uc3m | Universidad Carlos III de Madrid

Grado en Sistemas Audiovisuales
2016/2017

Trabajo de Fin de Grado

Modelos de mezclas de regresiones lineales para computación distribuida

Jorge Pereira Delgado

Tutora
Vanessa Gómez Verdejo

Agradecimientos

A mis padres, por su continua ayuda, comprensión, y por apoyarme en todas las decisiones que he tomado en mi vida, sin juzgarme cuando me he equivocado y ayudándome a levantarme sin reproches.

A mi tutora, Dr. Vanessa Gómez Verdejo, por su paciencia, calidad humana y por enseñarme un mundo nuevo cuya existencia no conocía y donde ahora me quiero seguir desarrollando.

A todos los miembros de la asociación juvenil BEST, por enseñarme que a la universidad se le puede sacar mucho más provecho del que parece a primera vista y donde he conocido personas excepcionales.

Y a todas las personas que influyen en mi día a día, porque gracias a ellos he llegado a ser lo que soy hoy en día.

Abstract

Big Data is a concept related to extremely large databases so that they cannot be processed with standard algorithms. For this reason, the concept of cluster computering was created. Here, the data is partitioned and processed in a computer cluster, so each one of the computers can process a part of the data and obtain a partial solution to our problem that, combined with the other partials solutions obtained from the other computer, we manage to obtain the solution. This way, we sharply reduce the computational cost of the original problem by breaking it into little subproblems.

This concept is quite new, so it is still developing. And here is where our bachelor's thesis makes the difference. The goal of this project is to develop an algorithm able to fit non linear regression problems, which is an underdeveloped field in the distributed machine learning branch. The only algorithms that can solve this kind of problems and are completely integrated in Spark's machine learning library [2] are the Random Forests and the Neural Networks; not even the Support Vector Machines are able to do it because there are no Kernel implementations integrated yet. Our model aims to solve this problem based on a Linear Regression Mixture Model.

The idea behind this is that, if we combine several Linear Regressions, we are able to break a non linear problem into several clusters that can be solved by linear algorithms. This way we manage to solve non linear problems and, furthermore, to do it in a more simple and interpretable way than other algorithms such as Random Forests or Neural Networks.

The first part of this bachelor's thesis will be to describe the different technologies and algorithms needed to fully understand the work developed here. Firstly, we explain all that we need to know about computer clusters and distributed computing. With a computer cluster we manage to have an excellent computing capacity without the need of a supercomputer. The problem here is that every computer in the cluster will only have a small part of the total data, thus not every algorithm is capable of being parallelized.

For an algorithm to be parallelizable, we have to check if we need all the data to obtain the result or if we can apply it to partitions of it. If we can apply it to small parts of the dataset, then our algorithm will be parallelizable. That will be an important point in our work, because some algorithms will not be able to be parallelized and we will have to use alternative, parallelizable algorithms.

In this part we also give a brief introduction to Apache Spark in its Python implementation, PySpark. We explain the most importants aspects of it. One important aspect of Spark is to understand the different kind of nodes in the computer clusters:

- Driver: Driver: manage the resources of the cluster and schedules all the tasks

to be performed.

- Workers: performs the different transformations and actions over the partitions of the dataset.

We also explain the RDDs or Resilient Distributed Databases, which are the basic parallelizable element in Apache Spark, and the two most important methods for the development of our algorithm in Apache Spark:

- map(): takes every data in the RDD and performs any function - passed as a parameter - we want individually. Usually, we will use a lambda function as the parameter. It returns a new RDD with the transformed data.
- reduce(): takes the data in the RDD and performs a binary, commutative and associative operation - passed as a parameter - to pairs of values and keeps the cumulative value in one of them, until it runs out of data and returns a single data, which will be returned to out driver.

After that, we introduce the gradient descent optimization based algorithms. The simplest one is the Gradient Descent. This is an iterative method that, for each step, gives a better solution of the problem until it converges to a local minimum. In each step, it computes the gradient of the loss function. The idea here is that the gradient gives us the direction in which the slope is steepest, so, if we move in the opposite direction of the gradient, we will end up in a local minimum. Here, the magnitude that we move, or step size, will be critical, because if the step size is too large the algorithm will diverge, and if it is too small it will take too long until it converges.

Once we have explained the basic gradient descent optimization algorithm, we are going to explain different evolutions that will fit better our problem and, in general, the distributed computing paradigm. Those algorithms are the Stochastic Gradient Descent and the Mini-Batch Gradient Descent. They take a small sample of the dataset - even only one in the case of the Stochastic Gradient Descent - and calculate a subgradient that will have the same expected value as the gradient. The subgradient will not be as precise as the gradient, but the computing time is much lower, which is critical when you are dealing with such big databases. Also, the Mini-Batch Gradient Descent, the one we will use here, usually converges very close to the point in which the standard Gradient Descent would have converged.

In the next part, we explain the different Machine Learning algorithms that will have relevance in the development of this bachelor‘s thesis. The first one are the Linear Regressions, which are based on, given a dataset, find a linear combination of a given set of functions, being the simplest case a line. To do it, we compute the minimum of the loss function, usually the Mean Squared Error or MSE.

Then, we explain a model that we will use as a starting point for our model: the Gaussian Mixture Models. These models assume that the data follow a given distribution, consisting on several Gaussians with different means and covariance matrices, and cluster the dataset. This is al iterative algorithm that consists in two steps:

- Expectation: for each point, and with the parameters of the distribution at that given time, it computes, for each sample, the probability of this sample of belonging to each one of the different Gaussian distributions, which we will call responsibilities.
- Maximization: assuming the responsibilities we computed previously, it estimates the parameters of the distribution - which are the means, covariance matrices and components sizes - such that the likelihood is maximized.

Everytime we repeat those two algorithms, we obtain a better solution to our problem, except in some cases if we are using Mini-Batch Gradient Descent and, specially, Stochastic Gradient Descent, but it will not be critical and only will occur few times. This algorithm is repeated until it converges to a local minimum or it exceeds a maximum number of iterations. This Expectation-Maximization algorithm, or EM, is the structure we will follow while developing our Linear Regression Mixture Models.

To finish with all the Machine Learning algorithms, we introduce the K-means algorithm. The K-means algorithm will be very important later, when we talk about possible methods to initialize the model. The K-means is a clustering algorithm, like the Gaussian Mixture Model. The K-means consists in finding the centroids for each cluster such that the total sum of the distances of the data points to it respective centroids is minimized. This algorithm also follows an EM structure.

In the last part of the State-of-the-art section, we explain a first approach to the Linear Regression Mixture Models. Although it is a very simple approach and it does not even talk about initializations or how to evaluate new data once our model is done, is an easy way to understand the basics of the algorithm and its EM structure.

The next part of this project consists in the algorithm development. To do this, we first implemented our own Gaussian Mixture Models in both, non distributed and distributed versions, to have a deeper understanding on the EM structure. After that, we implement our Linear Regression Mixture Model again in both, non distributed and distributed versions. This model consists of parameters that are a bit different of the ones in the Gaussian Mixture Models. Those are:

- Weights: weights of the linear regression for each component.

- Noise precision: inverse of the noise variance. It is a unique parameter for the whole model, since it is constant for every component.
- Component size: as in the Gaussian Mixture Models. The number of effective points in a component divided by the total number of data points.

With the implementation of the non distributed version, we try to have a better understanding of the problem and to face some problems that may have appeared - and, in fact, they did - in an easier and more well known scenario. After that, we implement the distributed version of the algorithm. Here is where the optimization algorithm comes in. In a non distributed algorithm, we would only need to compute the weights of the regression in a similar way as with the Moore-Penrose inverse - not exactly because here we have also the responsibilities in the equation - but this algorithm is not parallelizable, because we need to have all the data to compute it.

So here, instead of computing the weights with the inverse matrix, we optimize it with the Mini-Batch Gradient Descent Algorithm. This is easy to parallelize, because we can compute the contribution to the subgradient of each data point in the batch as partial results and combine them to get the whole subgradient. This way, we can optimize each one of the parameters of the distribution in the Maximization step.

Once we have a functional implementation of our model, we explore different parts that can make our algorithm better. We already have our model and it works, but, like the gradient descent based algorithms, our algorithm optimizes up to a local maximum, not the global one. This means that, using the same algorithm, we can have several different solutions, being some of them good and some of them bad. The reason it happens is because the minimum to which our algorithm converges depends on the starting point, it is, the initial value of the parameters of the distribution.

For that reason, one of the hardest part of this bachelor's thesis was to research and work on several initialization methods. Now, we will introduce the ones we decided to use to experiment with them:

- Initialization #1: using a K-means algorithm, we get N clusters, where N is the number of clusters out model will have. After that, we compute, for each group, a linear regression with the data points that belong to that cluster. The problem here is that we have to compute the weights of the linear regressions with a gradient descent based algorithm, so it will be slow and will not initialize to good values generally. To compute the size of the components, we only have to count the number of points in our cluster and divide by the total number of points, and the noise can be computed easily once we have the weights of the regression.

- Initialization #2: using a K-means algorithm, we get (N+1) clusters, where N is the number of clusters out model will have. This way, we have one extra centroid. Now, we can join the first centroid with the second, the second with the third and so on. For each pair, we have a line, so we can compute the N weights without performing a linear regression. Now we perform a K-means algorithm with N clusters and assign the weights to the new groups. Again, to compute the size of the components, we only have to count the number of points in our cluster and divide by the total number of points, and the noise can be computed easily once we have the weights of the regression.

- Initialization #3: using a K-means algorithm, we get 2N clusters, where N is the number of clusters out model will have. Now, we can pair the centroids by minimum distance in couples and compute the weights of the line they form. Again, to compute the size of the components, we only have to count the number of points in our cluster and divide by the total number of points, and the noise can be computed easily once we have the weights of the regression.

- Initialization #4: using a Gaussian Mixture Model, we get N components, where N is the number of components our model will have. Using the covariance matrices, we can get information about the preferred direction of the data in the cluster. With this direction and the mean of the cluster, we can compute the weights of the line. The noise can be computed, again, with values taken from the covariance matrices. The Gaussian Mixture Model return us the value of the components size, so we use that value to initialize it.

The other thing we can do to try to make our algorithm better is to be more selective when choosing the samples for the batch when we are optimizing the weights with the Mini-Batch Gradient Descent. Here, we take advantage of the fact that we do not only have the dataset, but also we have the responsibilities for each point. There are two things we have tried to improve the sampling method for the Mini-Batch Gradient Descent:

- Sampling method #1: when we are dealing with all the dataset to perform the gradient descent, there are data points that will produce a very small subgradient because of their responsibilities. If we first filter the responsibilities that are so low that will generate an insignificant subgradient, we can be sure that the points we sample will be relevant.

- Sampling method #2: we can assign a probability to each data point that is proportional to the responsibility for that component. This way, it will be easier to sample the data points with a higher responsibility, while it will be strange to pick the ones with low responsibilities. The problem here is that the batch size will be random too, although the mean value will be the one we assign.

Once we have developed all the model, including the initializations and sampling methods for the batch, we proceed to experiment with them, getting convergence times and Mean Squared Error or MSE as measurements. After that, we proceed to compare the different algorithms. Then, we present the conclusions we have extracted from those experiments. After that, we give a review of the future improvements and lines of research, which includes a deeper study on the different parameters of the algorithms and how them affect our model, an experimentation with a Big Data real problem, an improvement of the model and the creation of a ToolboX.

The last part of this report deals with the project planning and the project budget. In the first part, we talk about the different tasks that our project had and how they were distributed in time, describing each one of them. The second parts deals with the budgets. In one hand, we have the personal budgets and, in the other, the material costs.

Índice

Índice de figuras

Índice de tablas

1. Introducción

1.1. Descripción del problema

Big Data es un concepto que hace referencia a bases de datos tan grandes que las aplicaciones informáticas tradicionales de análisis de datos son incapaces de procesarlos. Por ello, surge el concepto de aprendizaje distribuido. En él, los datos se distribuyen en diferentes sistemas de almacenamiento y procesadores dentro de un clúster; de este modo, cada procesador trabaja en paralelo sobre un subconjunto de datos y, combinando las salidas parciales de cada procesador, se consigue así reducir la complejidad, coste y tiempo de computación en el tratamiento de los mismos [1].

Sin embargo, la implementación distribuida limita considerablemente el tipo de algoritmos de aprendizaje que pueden emplearse, ya que su formulación debe poder integrarse en el paradigma MapReduce. Así, por ejemplo, si nos fijamos en los algoritmos de clasificación y regresión disponibles en la librería MLlib [2], todos ellos utilizan implementaciones lineales basadas en el método de descenso por gradiente (Gradient Descent, GD).

El objetivo de este proyecto es conseguir una implementación distribuida de algoritmos de regresión no lineales. Para ello, utilizaremos modelos de mezclas de regresiones lineales (Linear Regression Mixture Models, LRMM) [3]. De este modo, resolveremos un problema no-lineal como una mezcla de problemas lineales, a la vez que ganamos en interpretabilidad en los resultados y lo hacemos con un algoritmo completamente integrado en el modelo MapReduce, en el cual se basa el desarrollo de Apache Spark.

En la Figura 1 se muestra un ejemplo de un problema con unos datos de entrada x y sus respectivos valores observados t. Vemos que estos no siguen una relación lineal, con lo que sería impensable ajustar dichos datos utilizando una regresión lineal para ajustar los datos de salida a los de entrada, ya que obtendríamos resultados inaceptables en cualquier aplicación. Sin embargo, si tomamos pequeñas porciones del conjunto de datos, vemos que dichos subconjuntos se pueden aproximar de forma bastante precisa con varias componentes, donde cada una de ellas es una regresión lineal diferente. Así, mezclando tres regresiones lineales, hemos resuelto de manera bastante satisfactoria un problema de regresión que no podía ser resuelto mediante una regresión lineal simple.

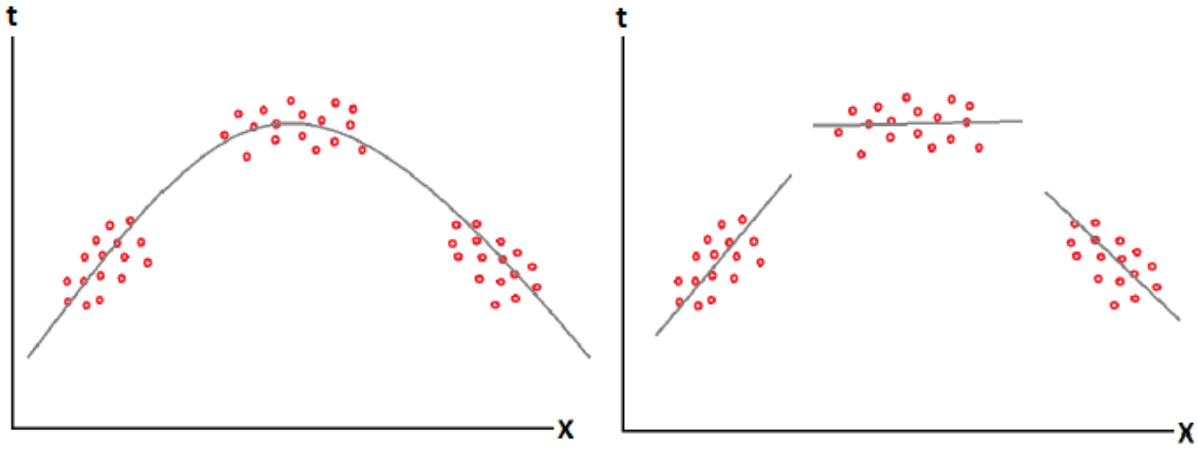


Figura 1: Ejemplo de problema no lineal y su respectiva solución mediante LRMM.

1.2. Motivación del trabajo de fin de grado

Existen hoy en día diferentes tipos de algoritmos que proporcionan soluciones a problemas de regresiones no lineales, y lo hacen con buenos resultados, como pueden ser regresiones basadas en redes neuronales (Neural Networks, NN), máquinas de vector soporte con Kernel (Suport Vector Machine, SVM) o bosques aleatorios (Random Forest, RF), sin embargo, dichos métodos presentan ciertos inconvenientes que conseguimos evitar con nuestro modelo y que se explican a continuación.

A favor de nuestro modelo, tenemos su simplicidad. Es muy sencillo entender el funcionamiento de una forma intuitiva, cosa que no ocurre en el caso de, por ejemplo, las NN, que requieren de un conocimiento mayor debido a la compleja estructura de las mismas o de las SVM, cuya formulación matemáticas es más complicada. Otra ventaja es la facilidad de interpretación del modelo. Al tomar nuevos datos de entrada, estos se comparan con los datos con los que hemos entrenado al modelo, eligiendo una mezcla de componentes con unos determinados pesos y obteniéndose un valor de salida que será nuestra predicción. Es fácil ver cómo afecta cada uno de los parámetros a nuestro LRMM. Sin embargo, en el caso de las NN, es prácticamente imposible saber cómo afecta un parámetro al valor de salida, debido a la gran cantidad de éstos que hay y a la enorme cantidad de interacciones producidas por las conexiones. En el caso de las SVM, estas sólo sirven para solucionar problemas no lineales si hacemos uso de algún Kernel, y al usarlos, perdemos toda la información que teníamos sobre los datos, ya que es como si estuviéramos trabajando con datos de una dimensionalidad mayor que no vemos, que en muchos casos es infinita. Además, en la librería MLlib, las SVM con Kernel aún no están integradas. En el caso de los RF, la aleatoriedad a la hora de construir cada uno de los arboles aleatorios (Random Tree), eligiendo al azar cada uno un subconjunto de muestras y de variables de los datos de entrada también dificulta la interpretabilidad del modelo.

Por otro lado, también es importante el coste de computación. Por ejemplo, entrenar una NN requiere de una gran cantidad de cálculos que hacen que dicho coste sea muy grande - obviando las NN sobre unidades de procesamiento gráfico, que agilizan notablemente su proceso de entrenamiento -, haciendo que estos modelos sean lentos de entrenar. Además, al ser nuestro modelo un modelo generativo, no tendrá el riesgo de sobreajuste que tienen los algoritmos anteriormente mencionados, y también nos servirá para generar nuevas muestras del modelo obtenido en caso de ser necesario.

Por estos motivos, además de por la sólida base sobre la que construir nuestro LRMM - los modelos de mezclas de Gaussianas (Gaussian Mixture Model, GMM) [4], que se describen en este trabajo -, consideramos este proyecto con potencial más que suficiente como para ser desarrollado.

1.3. Estructura del trabajo de fin de grado

El objetivo de este trabajo de fin de grado (TFG) es conseguir un algoritmo que se adapte a problemas de naturaleza no lineal y que éste sea capaz de ejecutarse en un clúster de ordenadores, es decir, que se adapte al modelo MapReduce. En el segundo capítulo, se explican las diferentes tecnologías y algoritmos, tanto de optimización como de aprendizaje máquina y computación distribuida, que se utilizarán a lo largo de este trabajo. También servirá para introducir algunas nociones básicas que serán importantes para entender mejor el desarrollo del TFG.

En el tercer capítulo nos centramos en el desarrollo del modelo. En una primera aproximación, realizamos una pequeña implementación no distribuida - esto es, que no podremos desarrollar posteriormente de forma eficiente en distribuido - que nos servirá como punto de partida para obtener un mejor entendimiento del problema y para encontrar posibles inconvenientes a los que nos podrá resultar beneficioso enfrentarnos en un escenario más sencillo y conocido para nosotros. Tras ello, se desarrolla el algoritmo distribuido, realizando los cambios pertinentes para sacarle el máximo partido a la computación distribuida. Una vez desarrollado el modelo, pasaremos a centrarnos en las diferentes inicializaciones del modelo, que resultan ser críticas en muchos casos a la hora de obtener un modelo válido para nuestro conjunto de datos, así como investigar diferentes métodos de muestreo y selección para nuestros algoritmos de optimización que podrían acelerar nuestro algoritmo y hacerlo más eficiente.

En el cuarto capítulo se presentan los resultados de los experimentos. En ellos, se comparan diferentes tipos de inicializaciones, así como diferentes tipos de algoritmos de optimización para el cálculo de los pesos de las diferentes regresiones lineales. Para ambos casos, se comparan tanto tiempos de convergencia como las prestaciones obtenidas. Estas últimas se comparan con los resultados obtenidos por una SVM con Kernel Gaussiano en su versión no distribuida - como ya hemos mencionado, no hay implementaciones de SVM con Kernel integradas en Spark aún.

En el quinto capítulo se exponen las conclusiones sacadas de los experimentos realizados una vez desarrollado el modelo. También se desarrolla brevemente qué posibles mejoras o líneas de desarrollo se podrían surgir sobre el trabajo realizado en este proyecto.

Por último, se estudia, por un lado, la planificación del proyecto, mostrando la evolución y la duración de cada una de las diferentes tareas en las que se ha dividido este TFG, y por otro lado, el presupuesto necesario para llevarlo a cabo, teniendo en cuenta tanto los costes personales como los materiales.

2. Estado del arte

En este capítulo se pasará a explicar algunos de los conceptos, tecnologías y técnicas más importantes que se usan en el ámbito de la computación distribuida y el aprendizaje automático (Machine Learning, ML), así como una revisión de los algoritmos GMM y su evolución a LRMM, que serán importantes para el desarrollo del TFG.

En la primera sección se explicará el concepto de computación distribuida, sus peculiaridades y por qué cobra especial importancia en el mundo del Big Data y el análisis y procesamiento de datos masivos. Tras esto, se pasa a describir los algoritmos de optimización de GD, descenso por gradiente estocástico (Stochastic Gradient Descent, SGD), y descenso por gradiente con mini-tandas (Mini-Batch Gradient Descent, MBGD), tratando de explicar su importancia para el desarrollo de algoritmos de ML en computación distribuida. Por último, se da una breve introducción al ML y se describen los algoritmos más relevantes para el desarrollo de este TFG, empezando por modelos de regresiones lineales simples, pasando por GMMs y explicando como éstos nos sirven como punto de partida para nuestro modelo, y acabando con una primera aproximación a los LRMM.

2.1. Computación distribuida: introducción a Apache Spark

En la actualidad, el análisis de datos masivos o Big Data en inglés está a la orden del día y su uso y desarrollo está en constante crecimiento. Sin embargo, en muchas ocasiones los algoritmos convencionales que utilizamos hacen que la cantidad de datos con la que podemos trabajar sea limitada. Esto se puede dar por dos motivos:

- El tiempo de computación crece demasiado rápido a medida que el tamaño del conjunto de datos crece.
- El espacio requerido en memoria es demasiado grande y no podemos almacenar los resultados obtenidos.

Por ello y por la imposibilidad de la mayoría de usuarios de trabajar con superordenadores, surgen los clústeres de ordenadores, los cuales son conjuntos de ordenadores que trabajan en una misma red como si fueran una computadora única. Es decir, se distribuyen las tareas que hay que llevar a cabo, trabajando varios ordenadores, máquinas virtuales o contenedores dedicados a esas tareas en paralelo.

De esta manera y gestionando cada nodo de manera correcta, conseguimos tener capacidades de cómputo mucho más grandes, enfrentándonos de manera independiente a diferentes fracciones del problema original en lugar de enfrentarnos al total. Una vez obtenido los resultados parciales en cada ordenador para cada

partición de los datos, éstos se combinan, obteniendo el resultado deseado en tiempos mucho menores que con métodos tradicionales.

En este trabajo se ha utilizado Apache Spark como framework para el desarrollo del mismo. Apache Spark se puede considerar un sistema de computación distribuida de propósito general y orientado a la velocidad. Además proporciona APIs en Java, Scala, Python y R, de los cuales hemos optado por utilizar el de Python debido al conocimiento previo de dicho lenguaje de programación, a su facilidad para el manejo de datos y su diversidad y flexibilidad de estructuras de datos.

Vamos ahora a tratar de explicar el funcionamiento de Apache Spark a la hora de repartir la carga dentro del clúster de ordenadores y cómo obtiene el resultado y qué limitaciones tiene esto. Por resumir, ya que no es objetivo de este TFG desarrollar este tema si no dar una breve introducción, cuando trabajamos con Apache Spark tendremos una máquina piloto o driver, que será la que ejecute nuestro aplicación de Apache Spark y el nodo más importante de nuestra red. Ésta partirá nuestra aplicación de Apache Spark en subtareas y las programará para ser ejecutadas en los trabajadores o workers, que son las máquinas que se encargaran de realizar las tareas pertinentes sobre su partición de los datos y devolver el resultado de la subtarea al driver.

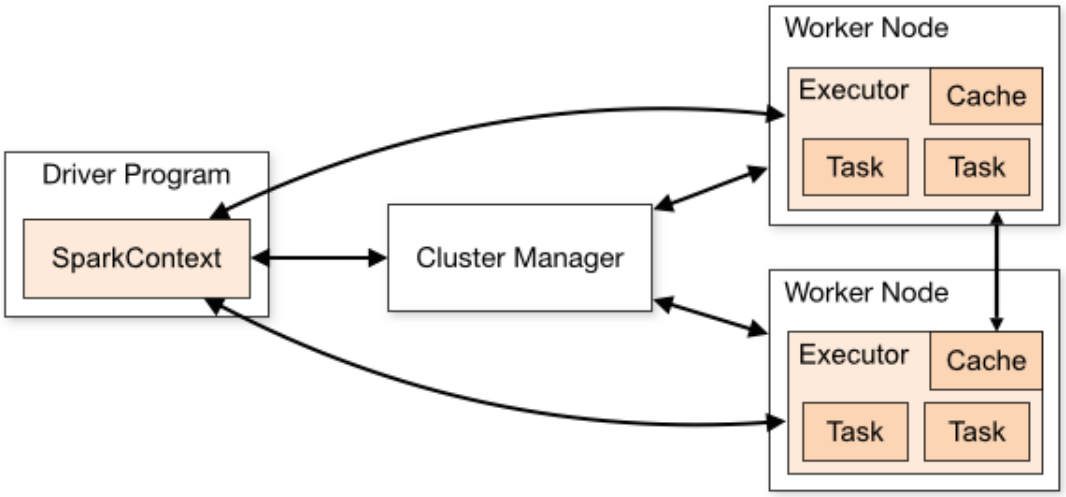


Figura 2: Modo de funcionamiento de un clúster de Apache Spark [7]

El driver actúa aquí como director de orquesta, balanceando la carga entre los diferentes trabajadores. Por ello, hemos se ser cautelosos a la hora de ejecutar nuestro programa, ya que éste podría verse saturado si ha de realizar alguna tarea concreta sobre el conjunto de datos, como una ordenación del mismo o la eliminación de datos duplicados, que producirían los mismos problemas que al ejecutar dichos algoritmos en una sola máquina si los conjuntos de datos son muy grandes.

Sin embargo, paralelizar un algoritmo no siempre es posible, ya que ha de adaptarse al la filosofía del MapReduce. Para saber si un algoritmo se puede implementar en distribuido o no, tenemos que fijarnos en si es necesario tener todos los datos para llevar a cabo el algoritmo o no. Por ejemplo, para calcular la media aritmética de un conjunto de datos no hace falta tener todos los datos; el primer trabajador podría calcular la suma de todos los datos de su partición, el segundo trabajador lo mismo con la suya y así para todos. Una vez calculadas las sumas, los resultados de éstas se le pasarían al driver, quien sólo tendría que sumar dichos resultados y dividir entre el número total de datos, con lo que vemos que el cálculo de la media es un algoritmo implementable en distribuido.

Por otro lado, otros algoritmos no lo son. Por ejemplo, si queremos llevar a cabo la inversión de una matriz necesitamos el conjunto de datos en su totalidad. Si llevamos a cabo las particiones y asignamos cada una de estas a un trabajador, estos no van a ser capaces de realizar el cálculo y obtener la matriz inversa. Por resumir, requerimos de todos los datos en un mismo trabajador para poder llevar a cabo esta tarea, lo que para conjuntos muy grandes de datos será imposible debido a las limitaciones de memoria y tiempo de computación que ya se han mencionado anteriormente en esta sección.

Esto cobra especial importancia en el TFG, como se explicará a lo largo del mismo, ya que el algoritmo que se utiliza en el modelo inicial para optimizar los pesos de las regresiones no es implementable en distribuido, y se ha de recurrir a otros que sí lo son para solventar este problema en nuestro modelo.

2.1.1. El paradigma MapReduce

Aquí merece la pena mencionar el modo de trabajo de Apache Spark para entender mejor cómo serán tratados nuestros datos a la hora de desarrollar nuestro modelo. Lo primero que hemos de mencionar, es que vamos a trabajar con Bases de Datos Resilientes Distribuidas (Resilient Distributed Datasets, RDDs). Éstos son las abstracciones básicas en Apache Spark, y representan una colección de elementos particionada e inmutable que puede ser tratada en paralelo. Operar sobre RDDs nos da una gran versatilidad debido a su nivel de abstracción y hay gran cantidad de métodos disponibles. Nos centraremos aquí en dos de ellos, que serán los más importanter para el desarrollo de nuestro modelo y que sirven para entender a grandes rasgos el funcionamiento de Apache Spark. Estos métodos son:

- *map*(): devuelve un nuevo RDD aplicando la función pasada por parámetro a cada elemento del RDD.
- *reduce*(): reduce los elementos de un RDD aplicando el operador binario, asociativo y conmutativo pasado como parámetro. Ya no devuelve un RDD, si no un dato del tipo del que fuera el RDD.

Un ejemplo sencillo y que muestra cómo se combinan estos dos métodos sería el siguiente. Supongamos que queremos contar el número de caracteres que tenemos en un texto. Lo primero que haríamos sería generar un RDD compuesto por las diferentes cadenas de caracteres que forman el texto para poder trabajar en distribuido. Una vez tenemos el RDD, generamos un nuevo RDD con ayuda de la funcion *map*() que aplique a cada elemento la función len(), teniendo ahora un RDD formado por números naturales con las longitudes de cada cadena de caracteres. Ahora, aplicando el método *reduce*(), podemos decir que, para cada par de datos, los sume y devuelva el resultado, manteniendo el valor acumulado y reduciendo así paulatinamente la cantidad de datos hasta tener uno sólo, que será el total de caracteres en el texto. A continuación, se muestra el código del ejemplo anterior.

```
myString = "Hola mundo!!!"

myStringRDD = sc.parallelize(myString)

countRDD = myStringRDD.map(lambda x : len(x))

totalCharacters = countRDD.reduce(lambda x, y : x + y)
```

Aquí vemos que nos creamos una cadena de caracteres en la primera línea de código y luego la paralelizamos con el método *parallelize*() del SparkContext. Tras esto, ya podemos trabajar en distribuido con el RDD obtenido. Primero, contamos el número de caracteres de cada partición con el método *map*(), utilizando una función anónima o lambda, y luego sumamos todos los resultados de las particiones con *reduce*(). Cabe destacar que podríamos juntar tantas operaciones sobre un RDD en una única línea de código como queramos, cosa que no hemos hecho con ánimo de explicar paso a paso el funcionamiento. Otra posibilidad, combinando el *map*() y el *reduce*() en una misma línea sería:

```
myString = "Hola mundo!!!"

myStringRDD = sc.parallelize(myString)

totalCharacters = myStringRDD.map(lambda x : len(x)).reduce(
   lambda x, y : x + y)
```

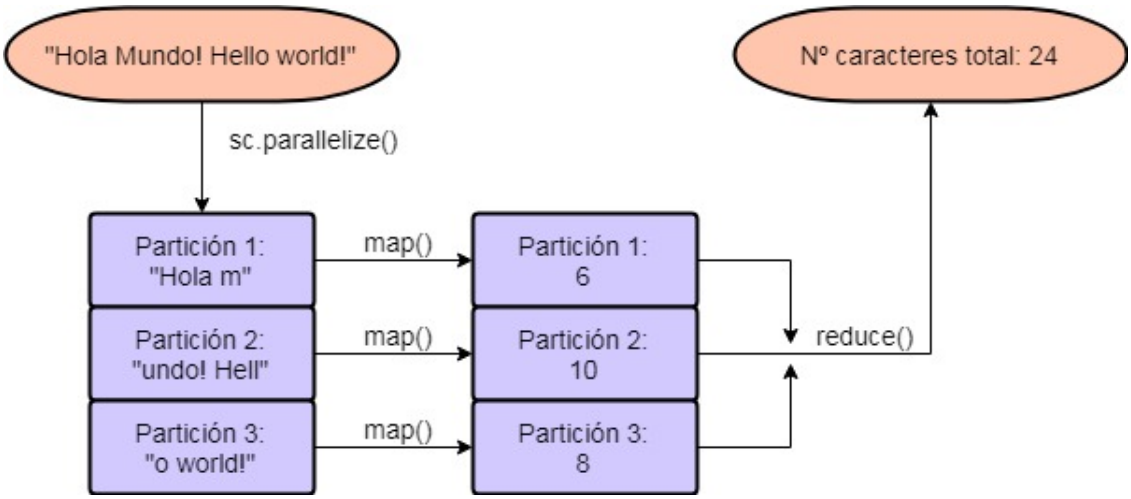


Figura 3: Esquema de funcionamiento de un algoritmo de Apache Spark con map-reduce.

Hay que tener ciertas cosas en cuenta al trabajar con RDDs. Como hemos dicho, los RDDs son estructuras que nos permiten trabajar en distribuido. Por ello, mientras trabajamos con ellos, los datos se encuentran distribuidos en los diferentes trabajadores de nuestro clúster de ordenadores, con lo que no será posible acceder a dichos datos desde nuestro nodo driver. Para ello, hemos de recolectar los datos de los trabajadores y que estén disponibles para nosotros. Esto se puede conseguir de varias formas. La más sencilla es aplicando el método *collect*() al RDD deseado, que devolverá todos los datos en los trabajadores. El problema es que todas y cada una de las particiones serán recolectadas, pudiendo saturar la memoria de nuestro driver. Si queremos obtener una pequeña porción de los datos para efectuar alguna comprobación, podemos usar el método *take*(), que nos devolverá tantos datos como queramos. El método *reduce*(), sin embargo, no devuelve un RDD, ya que reduce hasta sólo tener un único dato, que devuelve al driver, con lo que, tras llamar al método, sí tendremos el resultado disponible.

Cuando hemos definido el método *reduce*(), hemos comentado tres propiedades que tiene que cumplir el operador seleccionado: ha de ser binario, conmutativo y asociativo. La primera simplemente nos indica que toma los valores de dos en dos, y va acumulando el resultado, hasta que queden sólo dos valores que, al reducirlos una vez más, se quedan en uno, devolviéndose el valor al nodo driver.

Las otras dos propiedades tiene que ver con la estructura de los RDDs. Éstos no siguen un orden concreto, el programa puede particionar los datos de cualquier manera. Por ello, para que el resultado no dependa del orden de los datos, nuestro operador ha de tener las propiedades asociativa y conmutativa. Supongamos que tenemos tres valores, A, B y C. Como hemos explicado, el programa podría tomar cualquiera de las - en este caso - 6 operaciones posibles AB, AC, BA, BC, CA, CB como la primera a realizar. Si el operador seleccionado no fuera conmutativo, AB y BA - o cualquiera de las combinaciones simétricas - no darían el mismo resultado, haciendo que nuestro método *reduce*() resultara en diferentes resultados para

unos mismos datos de entrada. Lo mismo ocurriría si no se cumpliera la propiedad asociativa, ya que (A + B) + C sería diferente a A + (B + C), obteniendo el mismo problema.

2.2. Métodos de optimización basados en gradiente

Como se ha explicado en la Sección 2.1, no todos los algoritmos son aptos para trabajar con ellos de forma distribuida, debido a que el problema no se puede resolver una vez hechas las particiones sobre los datos. Por ello, a la hora de solventar este problema utilizaremos métodos de optimización basados en GD, ya que estos sí que se pueden paralelizar fácilmente, consiguiendo así un algoritmo que sí se pueda ejecutar de manera eficiente en un clúster de ordenadores y pueda lidiar con volúmenes de datos muy grandes en un tiempo razonable.

2.2.1. Descenso por gradiente

Los métodos basados en gradiente aprovechan la información que éste nos da sobre la función que queremos optimizar. El gradiente se puede interpretar como la dirección hacia la cual la pendiente es máxima para un punto dado. Así, sabemos que si nos movemos en esa dirección, el valor de la función se verá incrementado, mientras que si lo hacemos en dirección opuesta, éste se verá decrementado. No solo eso, si no que, en las proximidades de ese punto y siempre que la función sea continuamente diferenciable, esta será la dirección que provocará un incremento o decremento mayor al movernos en dicha dirección.

Así, para encontrar un máximo en una función, podemos, de forma iterativa, calcular el gradiente de ésta en el punto dado y movernos en esa dirección. Para minimizarla, nos movemos en dirección opuesta, llegando de esta forma a máximos y mínimos locales respectivamente, siempre y cuando el tamaño de los pasos que damos sea el adecuado.

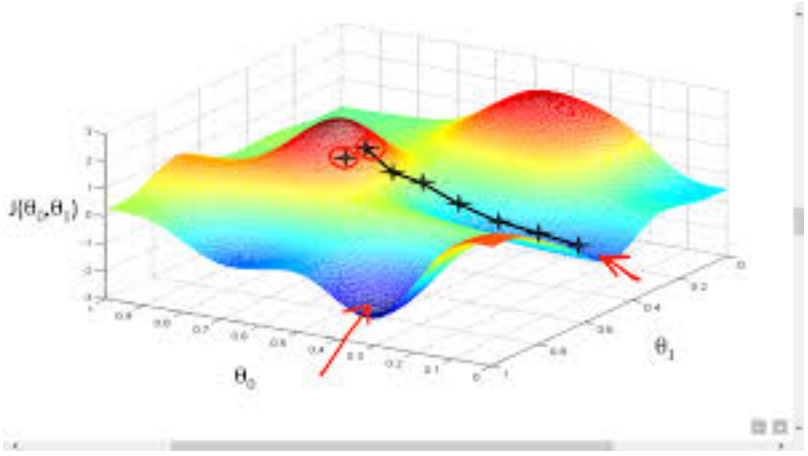


Figura 4: Ejemplo bidimensional del algoritmo GD [7]

Así, tenemos que el algoritmo GD calcula el gradiente de la función en el punto dado, obteniendo así la dirección en la que nos hemos de mover y una magnitud en función de lo empinada que esté la pendiente. En cada iteración, tomamos el antiguo valor de los parámetros y le restamos el gradiente que hemos obtenido, multiplicado por un tamaño del paso (Step Size, SS), moviéndonos así a un punto menos elevado, alcanzando finalmente un mínimo local. La formulación sería la siguiente

$$\omega^{n+1} = \omega^n - \gamma \nabla f(\omega^n) \tag{1}$$

donde ω son los parámetros a optimizar en la iteración dada por el superíndice, γ es el SS y f es la función a optimizar, siendo esta

$$f(\omega) = \lambda R(\omega) + \frac{1}{N} \sum_{i=1}^{N} C(\omega; x_i, y_i) \tag{2}$$

donde R es el término de regularización y C es la función de coste para el dato i-ésimo.

El método del GD es similar a liberar una canica en un punto dado de una superficie. La canica rodará en dirección opuesta al gradiente, es decir, cuesta abajo, y acabará depositándose en un valle, que sería el equivalente a un mínimo local, con la diferencia de que, en este símil, la canica estaría continuamente 'calculando' el gradiente.

Sin embargo, al tratar de optimizar funciones mediante este tipo de métodos nos encontramos con dos problemas:

- Convergencia del algoritmo: como hemos visto, es sencillo saber en qué dirección nos tenemos que mover a la hora de minimizar o maximizar una función. Sin embargo, la magnitud de ese cambio o SS, no es tan fácil. Una mala elección del SS en cada iteración puede llevar a un incorrecto funcionamiento del algoritmo. Si el SS es muy pequeño, el algoritmo tardará demasiado tiempo en converger o no lo hará, y si es demasiado grande podríamos acabar en un punto más alejado del mínimo del que ya estábamos, con lo que el algoritmo divergiría al ser la pendiente mayor a cada iteración, como se muestra en la Figura 2.2.1.

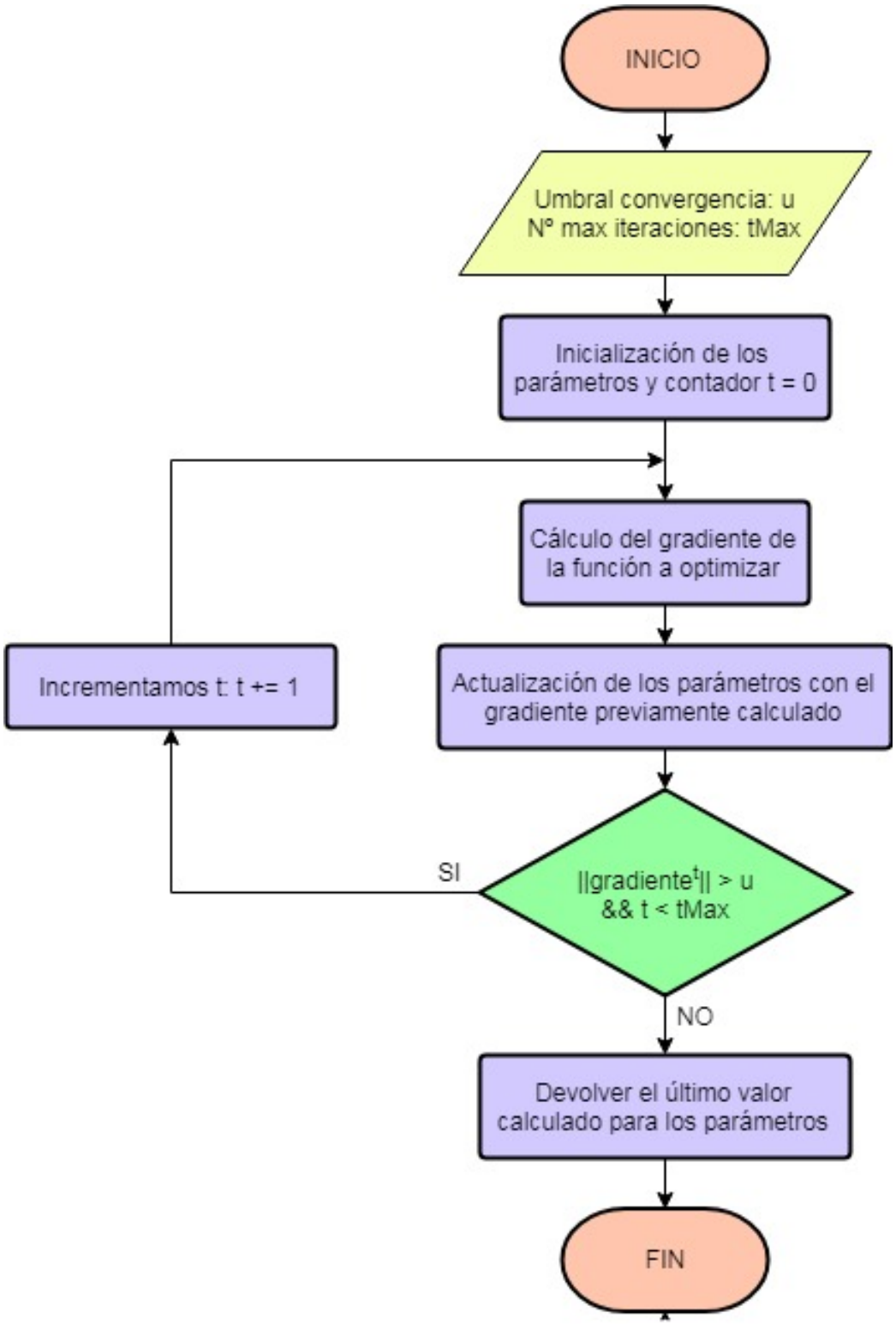


Figura 5: Diagrama de flujo del algoritmo GD.

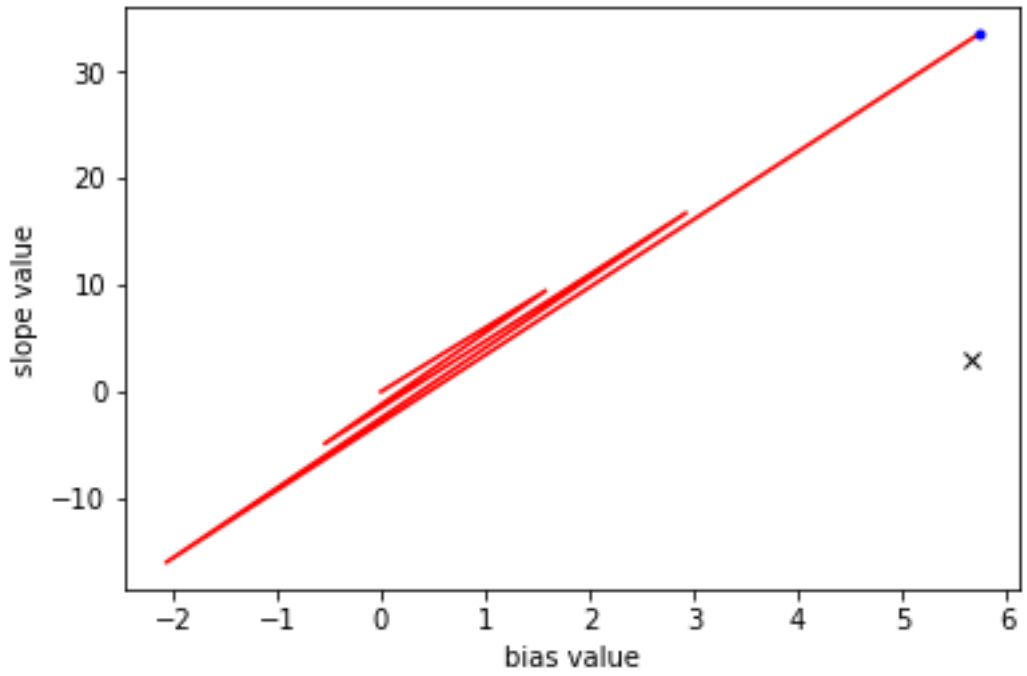


Figura 6: Ejemplo de divergencia del algoritmo GD

- Optimización local: esta familia de algoritmos sólo optimizan la función localmente. Esto quiere decir que no resuelven el problema de encontrar el mínimo absoluto, si no uno local. Además, este mínimo local que se toma como solución depende mucho de la inicialización. Volvamos al ejemplo de la canica sobre una superficie suave. Ahora, imaginemos que en un punto dado hay un máximo sobre la superficie, que sería como un monte en un paisaje, y a cada lado, un valle, de los cuales uno es más profundo que el otro. Si soltamos una canica en la cima del monte, pero ligeramente desplazada hacia un lado, acabará en uno de los valles, mientras que si el punto inicial de la canica estuviera ligeramente desplazado en la otra dirección, habría acabado en el otro. El algoritmo encontrará diferentes soluciones a un mismo problema por el simple hecho de que la inicialización de los parámetros, que se puede entender como el punto en el que la canica empieza a rodar, no son iguales, encontrando en un caso una solución mejor que en el otro.

Sin embargo, en ciertos tipos de problemas, como el que nos concierne en este trabajo, este último punto no es un problema, ya que la función de coste que usamos es continua, diferenciable y convexa, teniendo un único mínimo y siendo el gradiente no nulo para cualquier punto que no sea el mínimo.

2.2.2. Descenso por gradiente estocástico y por mini-tandas

No obstante, para calcular en cada iteración el gradiente, hemos de calcular el valor de la función de coste para cada dato, como podemos ver en la Ecuación (2), haciendo que el algoritmo sea más costoso cuanto mayor sea el conjunto de datos sobre el que estamos trabajando. Hay que tener en cuenta además que es un algoritmo iterativo, que puede repetirse cientos e incluso miles de veces antes de

converger.

Al estar tratando con conjuntos de datos tan grandes que se ha de trabajar con ellos de forma distribuida en un clúster de ordenadores, es de vital importancia reducir al máximo el coste computacional de estas operaciones. Por ello, utilizaremos métodos de optimización alternativos que simplifiquen y reduzcan dichos cálculos a la par que sigan entregando resultados que, si bien no son los óptimos, se aproximan lo suficiente como para que su uso sea beneficioso para nuestros objetivos.

Aunque se explicarán tanto el SGD como el MBGD, será el segundo el que utilicemos, como ya se verá más adelante. El SGD se basa en conseguir un subgradiente de manera aleatoria de tal forma que el valor esperado del subgradiente obtenido será igual al gradiente que resultaría de utilizar todos los puntos del conjunto de datos. Así, para cada iteración del algoritmo, en lugar de calcular el gradiente con todos los datos, simplemente muestrearemos uno. Así, la función a optimizar $f(\omega)$ quedaría de la forma

$$f_{est}(\omega) = \lambda R(\omega) + C(\omega; x_i, y_i) \tag{3}$$

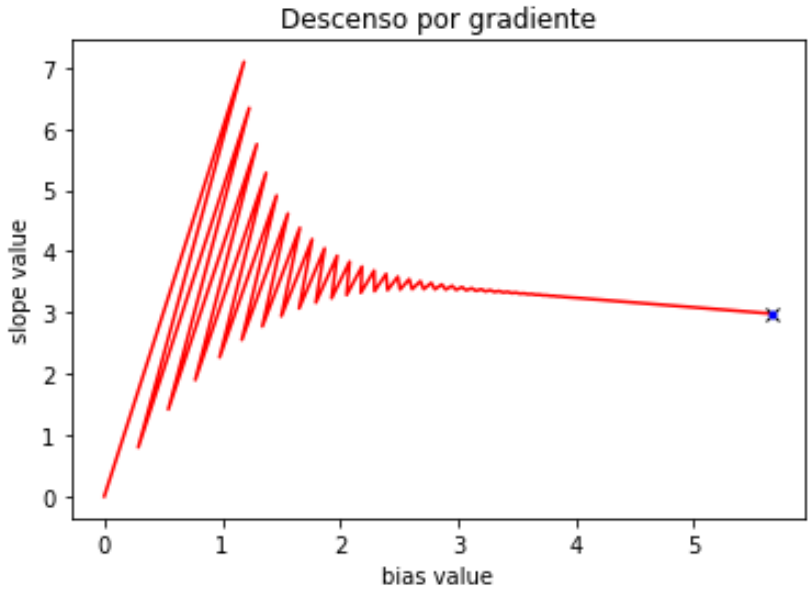
Así, el GD calcula el gradiente sobre el promedio de todos los costes, mientras que el SGD calcula un subgradiente utilizando una observación elegida al azar. Si enfocamos esto desde un punto de vista estadístico, nos damos cuenta de que tanto el gradiente como el subgradiente van a tener el mismo valor esperado. Si nos fijamos en las funciones que optimizamos en el método GD 2 y descenso por gradiente estocástico 3, vemos que

$$E[f(\omega)] = E[f_{est}(\omega)] = \mu_x \tag{4}$$

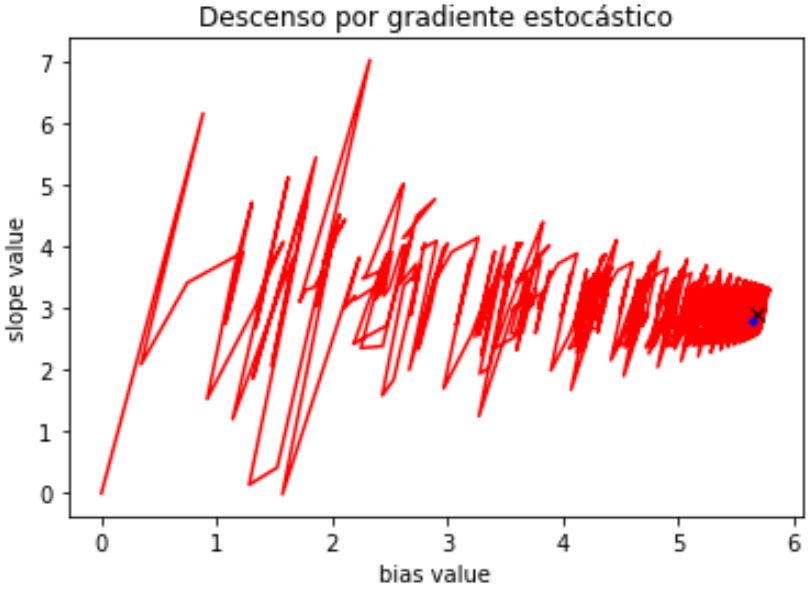
donde μ_x es la media de la distribución. Vemos así que E$[f(\omega)]$ y E$[f_{est}](\omega)$, se pueden entender como estimadores del gradiente medio y ambos son estimadores insesgados, con lo que, como ya habíamos apuntado, el valor esperado en ambos casos es el mismo. Esto hace que el algoritmo SGD converja también, aunque lo haga de manera menos precisa, acercándose en promedio al mínimo global.

El método de descenso de gradiente estocástico es, pues, computacionalmente menos costoso, ya que independientemente del tamaño del conjunto de datos utiliza sólo uno en cada iteración, a costa de ser menos preciso. En la Figura 7 se muestra la comparación entre el algoritmo GD tradicional y uno SGD.

Podemos observar que el método GD converge de forma más directa hacia el mínimo mientras que el método SGD lo hace de forma más irregular, llegando incluso a alejarse del mínimo en lugar de acercarse en algunas iteraciones. Esto se debe a que se utiliza una sola observación en cada iteración, lo que puede provocar



(a) El algoritmo GD converge de manera uniforme y converge con gran precisión



(b) El algoritmo SGD converge de manera más caótica y no converge de manera tan precisa.

Figura 7: Ejemplo de convergencia de los algoritmos GD y SGD.

que el subgradiente no se parezca en nada al gradiente que resultaría de usar todas las observaciones y promediar, ya que lo que nos asegura la versión estocástica del algoritmo es que el subgradiente será el mismo en promedio, no para cada observación.

Esto nos lleva a otro problema, y es que para ciertas observaciones, el subgradiente resultante puede ser demasiado grande y desplazarnos a un punto más alejado del mínimo global que estamos buscando, e incluso desplazarnos de él una vez lo hayamos encontrado, ya que aun encontrándonos con la solución óptima, esto es, el mínimo global, esa solución no será la óptima para una observación aislada, si no que lo es para el promedio de todas. Para solucionar esto se pueden tomar varios enfoques, como pueden ser:

- Elegir un SS lo suficientemente pequeño como para que las actualizaciones de los pesos también lo sean y no nos desplacemos demasiado una vez alcanzado un mínimo.
- Para cada iteración sobre todo el conjunto de datos, reducir el SS. Así, damos pasos más grandes al principio y, una vez nos acercamos al mínimo, vamos reduciendo el tamaño, aumentando la precisión del algoritmo.

Las implementaciones de la librería MLlib de Spark utilizan esta segunda opción, quedando las actualizaciones de los pesos como se muestra en la siguiente fórmula

$$\omega^{n+1} = \omega^n - \frac{\gamma}{\sqrt{t}} \nabla f(\omega^n, \phi(x_i)) \tag{5}$$

donde t es el número de iteraciones sobre el conjunto de datos que llevamos. Así, las primeras iteraciones daremos pasos más grandes, acercándonos de forma más rápida al mínimo, y más pequeños según avanzamos, afinando más sobre el mínimo y sin dar pasos demasiado grandes que podrían hacer desplazarnos de la solución una vez alcanzada.

Un término medio a estos dos algoritmos sería el MBGD. Este toma una muestra de determinado tamaño del conjunto de datos total y calculan el subgradiente a partir de ésta. Así, este algoritmo tendrá como ventajas un menor tiempo de computación que el algoritmo GD tradicional y mayor precisión que el SGD. En secciones posteriores se discutirán varias formas de muestreo a la hora de seleccionar los datos para su ejecución.

Ésta variante del algoritmo adquirirá además especial importancia en nuestro desarrollo del modelo. Al trabajar en un clúster de ordenadores, podremos realizar una operación simultáneamente por cada procesador que tengamos. Así, si hacemos coincidir el tamaño de la muestra a tomar con el número de procesadores, podremos llevar a cabo cada iteración del algoritmo en un tiempo igual - o

prácticamente igual - al que requeriría el SGD, obteniendo además un subgradiente estadísticamente más similar al gradiente total que el resultante del SGD.

2.3. Aprendizaje automático

En esta sección se hará un breve repaso de algunos algoritmos de aprendizaje máquina que serán utilizados de forma recurrente a lo largo del desarrollo de este trabajo, como las regresiones lineales, así como otros que nos servirán de punto de ayuda para el desarrollo del mismo, como los GMM o una primera aproximación a los LRMM en su versión no distribuida.

2.3.1. Regresión lineal

Uno de los algoritmos más sencillos cuando hablamos de aprendizaje máquina son las regresiones lineales. Se trata de algoritmos de aprendizaje supervisado, esto es, para cada dato tenemos un par de vectores de los cuales uno son los datos de entrada y el otro los de salida. Estos datos de salida serán valores reales.

Pasaremos ahora a explicar las regresiones lineales. Dado un conjunto de N observaciones $\{x_n\}$, donde $n = 1, ..., N$, junto con sus correspondientes valores de salida $\{t_n\}$, el objetivo es ser capaces de predecir el nuevo valor t para una nueva observación x. Supongamos el caso más sencillo, en el que por cada observación tenemos un dato de entrada y otro de salida. Suponemos así que el valor de salida para cada observación es función del valor de entrada x y los pesos ω, además de un ruido ϵ, teniendo que

$$t = y(x, \omega) + \epsilon \tag{6}$$

siendo ϵ un ruido aditivo Gaussiano de media nula y precisión - la inversa de la varianza - β. Así, podemos reescribir la fórmula de arriba como

$$p(t|x, \omega, \beta) = y(x, \omega) + \mathcal{N}(t|0, \beta^{-1}) = \mathcal{N}(t|y(x, \omega), \beta^{-1}) \tag{7}$$

Como lo que nos interesa es ajustar los pesos de una manera óptima a las observaciones y no tanto buscar la distribución exacta, en lugar de tratar de estimar $y(x, \omega)$, trataremos de buscar los pesos ω y la precisión del ruido β que hagan que la verosimilitud de nuestros datos sea máxima, la cual, al ser observaciones independientes e idénticamente distribuidas, se calcula como

$$p(t|x, \omega, \beta) = \prod_{n=1}^{N} \mathcal{N}(t|\omega^T \phi(x), \beta^{-1}) \tag{8}$$

donde $\phi(x)$ es una combinación lineal de diferentes funciones de x más un termino independiente. Sin embargo, como lo que nos interesa no es el valor de verosimilitud máximo si no dónde se encuentra este, podemos utilizar en su lugar la logverosimilitud, ya que es una función estrictamente creciente y los máximos no cambiarán de sitio, obteniendo

$$\ln p(t|x,\omega,\beta) = \sum_{n=1}^{N} \ln \mathcal{N}(t|\omega^T\phi(x),\beta^{-1}) = \frac{N}{2}\ln\beta - \frac{N}{2}\ln(2\pi) - \beta E_D(\omega) \quad (9)$$

donde $E_D(\omega)$ es la función suma de los errores cuadráticos y se define como

$$E_D(\omega) = \frac{1}{2}\sum_{n=1}^{N}[t_n - \omega^T\phi(x_n)]^2 \quad (10)$$

Vemos ahora que buscar los parámetros ω y β que maximizan la verosimilitud se resume a un simple problema de optimización. Empezaremos con la optimización respecto a ω. Si calculamos el gradiente, obtenemos que

$$\nabla \ln p(t|\omega,\beta) = \beta\sum_{n=1}^{N}[\omega^T\phi(x_n) - t_n]\phi(x_n) \quad (11)$$

Fijando el valor a cero y resolviendo, tenemos que

$$0 = \sum_{n=1}^{N} t_n\phi(x_n)^T - \omega^T\sum_{n=1}^{N}\phi(x_n)\phi(x_n)^T \quad (12)$$

que podemos reescribir en forma matricial y, despejando ω, obtenemos finalmente que

$$\omega_{ML} = (\Phi^T\Phi)^{-1}\Phi^T t \quad (13)$$

donde ω_{ML} es el vector de pesos, t el de valores de salida de todas las observaciones y Φ la matriz que tiene por filas los valores de las observaciones y por columnas las funciones que se aplican a dichos valores.

También podemos maximizar la verosimilitud respecto a la precisión del ruido β. Derivando sobre β e igualando a 0, obtenemos

$$\frac{\partial}{\partial\beta}\ln p(t|\omega,\beta) = \frac{N}{2\beta_{ML}} - \frac{1}{2}\sum_{n=1}^{N}[t_n - \omega^T\phi(x_n)]^2 = 0 \quad (14)$$

que despejando nos queda como

$$\frac{1}{\beta_{ML}} = \frac{1}{N}\sum_{n=1}^{N}[t_n - \omega^T\phi(x_n)]^2 \tag{15}$$

que es calcular la varianza del ruido, como podíamos haber intuido.

2.3.2. Modelos de mezclas de Gaussianas

Los GMM son algoritmos de agrupamiento que se basan en suponer que los datos han sido generados por una distribución formada por una combinación lineal de componentes, todas ellas siguiendo su propia distribución Gaussiana, con su media y matriz de covarianzas. Así, podemos formular la distribución de la siguiente manera

$$p(x) = \sum_{k=1}^{K}\pi_k\mathcal{N}(x|\mu_k, \Sigma_k) \tag{16}$$

donde μ_k y Σ_k son el vector de medias y la matriz de covarianzas de la componente k-ésima respectivamente y π_k se puede entender como el tamaño relativo de dicha componente respecto al total. Si nos fijamos, para que dicha formula sea una función de densidad de probabilidad, tiene que cumplirse que

$$\sum_{k=1}^{K}\pi_k = 1 \tag{17}$$

donde

$$0 \leq \pi_k \leq 1 \tag{18}$$

De ahí que hablemos del tamaño relativo de la componente, ya que es la proporción de muestras de dicha componente con respecto al total.

Para comprender correctamente cómo funciona el algoritmo, es necesario introducir el concepto de responsabilidad $\gamma(z_k)$, que se puede entender como la probabilidad que tiene una observación dada de haber sido producida por la componente k-ésima de la distribución. El cálculo de la responsabilidad es sencillo. Primero, calculamos la verosimilitud de la observación dada para cada una de las componentes y las sumamos. La responsabilidad para una componente dada será el cociente entre la verosimilitud para esa componente y la suma de todas ellas.

$$\gamma(z_k) = p(x \in \text{clúster } k) = \frac{\pi_k\mathcal{N}(x|\mu_k, \Sigma_k)}{\sum_{j=1}^{K}\pi_j\mathcal{N}(x|\mu_j, \Sigma_j)} \tag{19}$$

Dejaremos por ahora de lado el concepto de responsabilidades, que volverá a aparecer más adelante, y continuaremos ahora con el desarrollo de los GMM. Una vez conocemos la función de densidad, el problema vuelve a enfocarse en la optimización de los diferentes parámetros que tenemos para conseguir que la verosimilitud sea máxima. Aplicando el mismo truco que para pasar de la Ecuación 8 a la 9, obtenemos la siguiente función a optimizar a partir de la Ecuación 16

$$\ln p(x|\pi,\mu,\Sigma) = \sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{K} \pi_k \mathcal{N}(x_n|\mu_k,\Sigma_k) \right\} \tag{20}$$

Sin embargo, este tipo de problemas no es tan sencillo de optimizar como las regresiones lineales vistas en la Sección 2.3.1 o como en el caso de una sola Gaussiana. Esto se debe al sumatorio sobre k que aparece dentro del logaritmo. Vamos a ver por qué es un problema tan complicado. Para ello, vamos a tratar de optimizar el parámetro μ. Para ello, procedemos a calcular la derivada con respecto a μ e igualar a cero.

$$\frac{\partial}{\partial \mu_k} \ln p(x|\pi,\mu,\Sigma) = -\sum_{n=1}^{N} \frac{\pi_k \mathcal{N}(x_n|\mu_k,\Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(x_n|\mu_j,\Sigma_j)} \Sigma_k (x_n - \mu_k) \tag{21}$$

que sustituyendo con la expresión 19 e igualando a 0, nos queda

$$\frac{\partial}{\partial \mu_k} \ln p(x|\pi,\mu,\Sigma) = -\sum_{n=1}^{N} \gamma_{nk} \Sigma_k (x_n - \mu_k) = 0 \tag{22}$$

Como vemos, no podemos optimizar la función para el parámetro μ, ya que el propio parámetro influye en el cálculo de las responsabilidades. Hay que resaltar que esto ocurre con los parámetros π y Σ también. Una forma elegante y sencilla para resolver problemas de este tipo, en los que una variable latente hace muy complicada la optimización, es utilizar el algoritmo *expectation − maximization* (EM) [11].

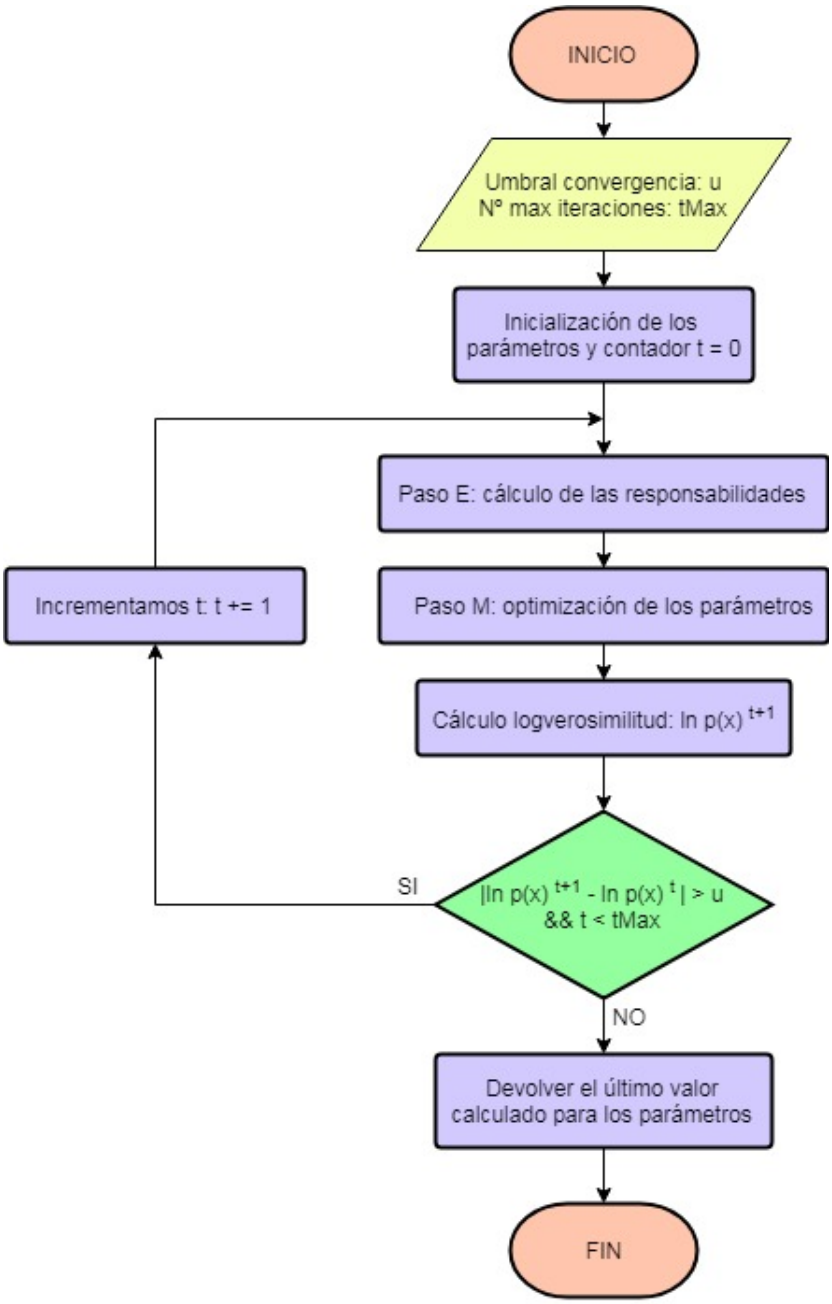


Figura 8: Diagrama de flujo del algoritmo EM.

Este método es un método iterativo que consta de dos pasos y que, como veremos más adelante, aumenta la verosimilitud en cada iteración, consiguiendo una mejor solución cada vez. El primero, el paso *expectation* (E) consiste en calcular el valor esperado de la verosimilitud - utilizaremos la logverosimilitud en nuestro caso - y las responsabilidades en función de la distribución condicional de las observaciones respecto a los parámetros y al valor de nuestras estimaciones de éstos.

El paso *maximization* (M) toma las responsabilidades obtenidas del paso anterior y maximiza la verosimilitud con respecto a los diferentes parámetros. Como vemos, aquí no se está teniendo en cuenta que las responsabilidades cambiarán al

cambiar el valor de los parámetros. Ahora entra en juego el hecho de que sea un algoritmo iterativo. Tras el paso M, le sigue otra vez el paso E. En él, se vuelven a calcular las responsabilidades y la logverosimilitud con los parámetros nuevos, que se volverán a actualizar en el siguiente paso M. Así, iterativamente, vamos calculando las responsabilidades y maximizando los parámetros iterativamente, obteniendo cada vez mejores soluciones hasta que se converge a un mínimo local.

Estudiaremos ahora la optimización en el paso M para los diferentes parámetros, empezando por las medias μ, ya que ya hemos obtenido la derivada en la Ecuación 22. Despejando la μ_k de ella, obtenemos que

$$\mu_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma_{nk} x_n \tag{23}$$

donde N_k la hemos definido como

$$N_k = \sum_{n=1}^{N} \gamma_{nk} \tag{24}$$

y que puede interpretarse como la cantidad de puntos efectivos en un grupo k dado o, lo que es lo mismo, el tamaño de dicho grupo. Aplicando el mismo desarrollo para la matriz de covarianzas Σ_k, obtenemos el siguiente resultado

$$\Sigma_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma_{nk} (x_n - \mu_k)(x_n - \mu_k)^T \tag{25}$$

Por último, para maximizar respecto a π_k tenemos que tener en cuenta que tenemos la restricción 17. Aplicando el método de los multiplicadores de Lagrange, tenemos que la función a optimizar es

$$\ln p(X|\pi, \mu\Sigma) + \lambda\Big(\sum_{k=1}^{K} \pi_k - 1\Big) \tag{26}$$

que derivando da como resultado

$$0 = \sum_{n=1}^{N} \frac{\mathcal{N}(x_n|\pi_k, \mu_k, \Sigma_k)}{\sum_j \pi_j \mathcal{N}(x_n|\pi_j, \mu_j, \Sigma_j)} + \lambda \tag{27}$$

Despejando el sistema de dos ecuaciones y dos incógnitas formado por esta ecuación y por la ecuación 17, tenemos que $\lambda = -N$ y que

$$\pi_k = \frac{N_k}{N} \tag{28}$$

Así, vemos que el parámetro π es un vector que se puede interpretar como el tamaño relativo de cada grupo respecto al total, como ya habíamos apuntado en esta sección anteriormente.

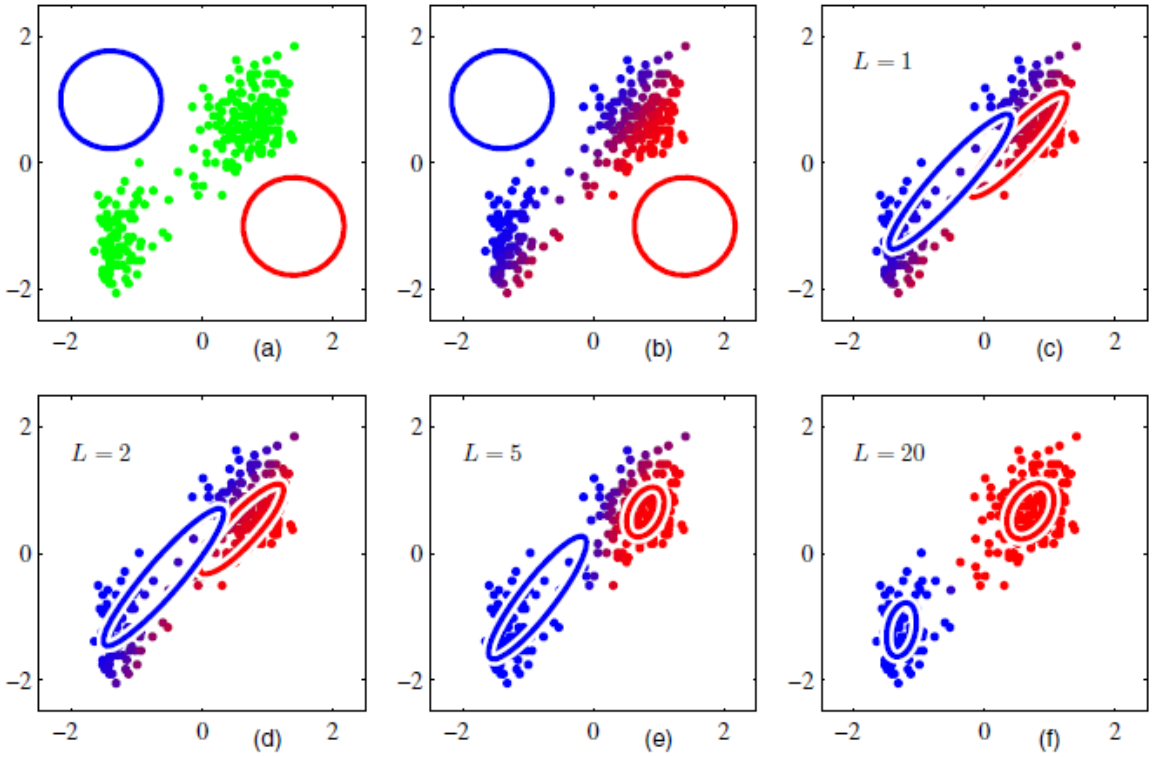


Figura 9: Ejemplo de evolución de un modelo GMM con dos componentes [3].

Dicho algoritmo se repite hasta alcanzar un punto en el que la logverosimilitud aumenta por debajo de un umbral marcado o el número de iteraciones supera un máximo, en cuyo caso podemos suponer que la inicialización no ha sido buena. Hay que tener en cuenta que este algoritmo no asegura converger al mínimo absoluto, con lo que la inicialización es crítica.

2.3.3. Mezclas de modelos de regresiones lineales

A partir del GMM estudiado, resulta fácil seguir un desarrollo similar para conseguir un LRMM basándonos nuevamente en el algoritmo EM, pero adaptándolo a los parámetros del problema. Si, al igual que hemos hecho en la Sección 2.3.2, extendiendo la ecuación de la función de densidad de probabilidad de una Gaussiana simple a una combinación lineal de varias componentes, lo hacemos con una regresión lineal y la extendemos a varias componentes, obtenemos la siguiente fórmula

$$p(t|\theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(t|\omega_k^T \phi, \beta^{-1}) \tag{29}$$

Donde θ es el conjunto de parámetros que hemos de ajustar, y éstos serían:

- ω: el valor de los pesos para cada una de las diferentes regresiones lineales que forman el modelo.
- π: igual que en el caso de los GMM. Indica el tamaño relativo de la componente respecto al tamaño total del conjunto de datos.
- β: precisión del ruido, o lo que es lo mismo, el inverso de la varianza. En este caso, supondremos que es un valor único para todas las componentes del modelo.

Aplicando logaritmos y desarrollando como hemos hecho en la Sección 2.3.2, tenemos que la logverosimilitud del modelo quedaría como

$$\ln p(t|\theta) = \sum_{n=1}^{N} \ln \Big(\sum_{k=1}^{K} \pi_k \mathcal{N}(t|\omega_k^T \phi(x_n), \beta^{-1}) \Big) \tag{30}$$

y además, vemos que el concepto de responsabilidad aquí vuelve a ser equivalente, y lo único que tenemos que hacer es cambiar la forma de calcularlo para que esté acorde a la función de densidad de probabilidad que hemos definido. Así, las responsabilidades vendrían dadas por

$$\gamma_{nk} = p(k|\phi(x_n), \theta^{old}) = \frac{\pi_k \mathcal{N}(t_n|\omega_k^T \phi(x_n), \beta^{-1})}{\sum_j \pi_j \mathcal{N}(t_n|\omega_j^T \phi(x_n), \beta^{-1})} \tag{31}$$

que de nuevo, aparecerá a la hora de optimizar los diferentes parámetros en el paso M del algoritmo EM. Tras esto, pasamos a optimizar los diferentes parámetros de la misma forma que en la Sección 2.3.2. Si empezamos por π, obtenemos el mismo resultado que en la Ecuación 24.

$$\pi_k = \frac{1}{N} \sum_{n=1}^{N} \gamma_{nk} = \frac{N_k}{N} \tag{32}$$

Para el parámetro β el resultado es similar a 25, pero aquí el ruido es unidimensional en lugar de ser una matriz de covarianzas y se utiliza la diferencia al cuadrado entre el valor de la regresión, que vendrá dado por el producto escalar $\omega_k^T \phi(x_n)$, y el valor observado t_n para calcular la precisión del ruido

$$\frac{1}{\beta} = \frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\gamma_{nk}(t_n - \omega_k^T\phi(x_n))^2 \tag{33}$$

Por último, queda la optimización de los pesos para cada una de las componentes. Como podíamos suponer, el resultado guardará ciertas similitudes con la solución 13, solución para el caso de una regresión lineal simple. Así, derivando respecto a los pesos e igualando a cero, tenemos que:

$$0 = \sum_{n=1}^{N}\gamma_{nk}(t_n - \omega_k^T\phi_n)\phi_n \tag{34}$$

que se puede reescribir de forma matricial de la siguiente manera

$$0 = \Phi^T R_k(t - \Phi\omega_k) \tag{35}$$

donde $R_k = diag(\gamma_{nk})$ es una matriz diagonal de dimensión $N \times N$ en la que, para la posición n-ésima de la diagonal, tenemos el valor n-ésimo del vector de responsabilidades para la componente k-ésima. Despejando ω_k, obtenemos que

$$\omega_k = (\Phi^T R_k \Phi)^{-1}\Phi^T R_k t \tag{36}$$

que vemos que es un resultado muy similar al obtenido al resolver una regresión lineal simple, con la diferencia de que tenemos la matriz R_k que pondera el efecto que cada observación tendrá sobre cada componente en función de lo probable que sea que dicha observación haya sido generada por esa componente. Al igual que para el caso de GMM, este algoritmo se itera hasta que converge a un mínimo, sin poder asegurar que éste vaya a ser global.

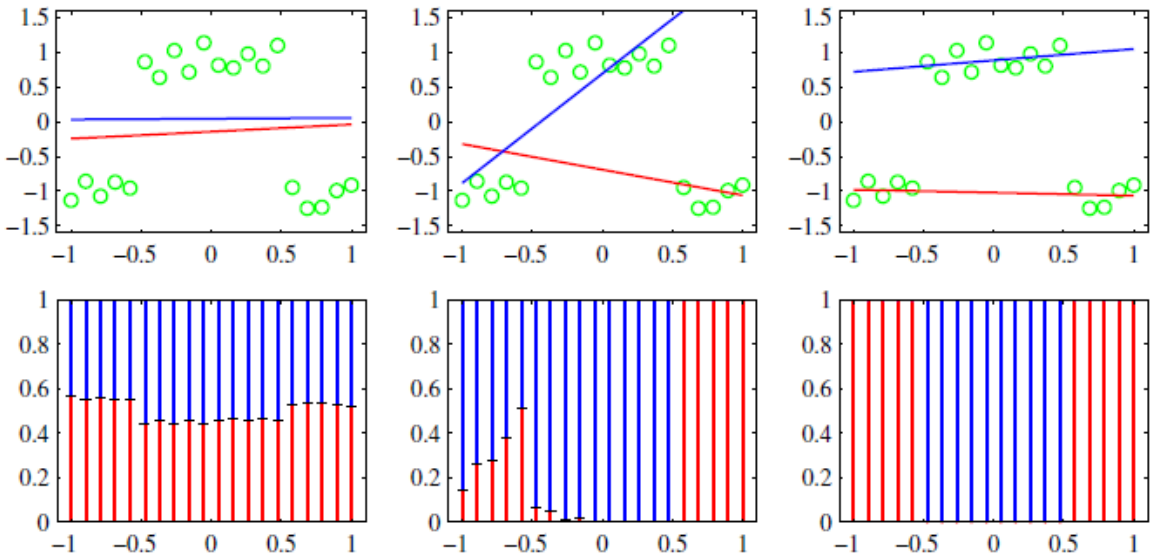


Figura 10: Ejemplo de evolución de un LRMM con dos componentes. Arriba se muestra la evolución de las regresiones, abajo, las responsabilidades de cada punto para cada componente [3].

2.4. K-medias

Se ha decidido dedicar una pequeña sección al algoritmo K-medias, que es un algoritmo de agrupamiento que minimiza la suma total de las distancias de cada punto al centroide de su grupo correspondiente. Esto se debe a que la inicialización es una parte muy importante de este trabajo, y este algoritmo cobra mucha importancia a la hora de desarrollar ciertos algoritmos de inicialización. Recordemos que nuestro LRMM se basa en una estructura EM que hace que a cada iteración obtengamos un resultado mejor que el anterior. Sin embargo, este tipo de algoritmos proporcionan resultados óptimos locales, no absolutos, de aquí que la inicialización sea una parte tan importante del trabajo. Además, como explicaremos a continuación, el algoritmo de K-medias se puede integrar perfectamente en el paradigma de Spark y es completamente funcional en distribuido.

El algoritmo de K-medias trata de minimizar la suma total de las distancias de cada punto del conjunto de datos al centroide de su grupo, donde un punto pertenece a un grupo dado si el centroide de dicho grupo es el más cercano a dicho punto. En definitiva, se trata de un problema de optimización del parámetro

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} ||x_n - \mu_k||^2 \tag{37}$$

donde r_{nk} es una variable que valdrá 1 en caso de que el dato pertenezca al grupo k-ésimo y 0 en caso de pertenecer a otro, μ_k es el centroide del grupo k-ésimo y

J es la suma de las distancias totales de cada punto a su centroide correspondiente.

Dicho algoritmo también sigue una estructura similar a la EM, con dos pasos que se siguen de forma iterativa hasta que el algoritmo converge. Los pasos son:

- Paso de asignación: cada punto se asigna a uno de los grupos. Un punto se asigna a un grupo si la distancia a su centroide es menor que al resto de centroides.
- Paso de actualización: se calculan los nuevos centroides para cada grupo. Estos se computan como la media de los puntos que pertenecen a dicho grupo, como se muestra a continuación

$$\mu_k^{t+1} = \frac{1}{N_k} \sum_{n=1}^{N} r_{nk} x_n \tag{38}$$

  donde N_k es la cantidad de puntos asignados al grupo k-ésimo.

Así, iterativamente se van consiguiendo centroides que minimizan la distancia a los mismos de los miembros de su grupo. Si en los otros modelos presentados aquí esto se veía mostrado en un incremento de la logverosimilitud de los datos, en este caso se observa una reducción de la distancia total J a cada paso de la iteración. Cabe mencionar que este algoritmo cuenta con una gran complejidad computacional, siendo un problema de tipo NP-duro. Sin embargo, hay un gran trabajo detrás de las implementaciones actuales del algoritmo y cuentan con heurísticas que han mostrado ser bastante efectivas y que hacen que la convergencia al mínimo local sea rápido. Sin embargo, no es objetivo de este trabajo centrarnos en ellas, ya que el algoritmo k-medias sólo entra en juego en la inicialización de nuestros parámetros, y basta con tener una ligera idea de su funcionamiento.

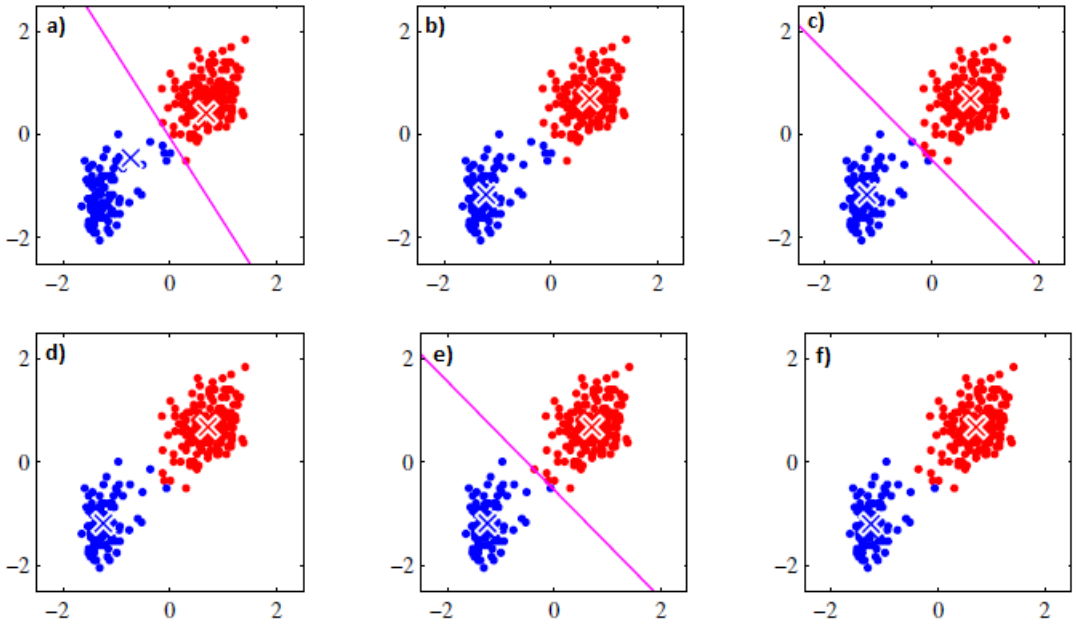


Figura 11: Ejemplo de la evolución de un algoritmo de k-medias con 2 componentes [3].

3. Desarrollo del modelo

3.1. Implementación del algoritmo no distribuido

Con el fin de entender mejor el problema y construir sobre una base lo más sólida posible el desarrollo de nuestro LRMM que sea a su vez distribuido, hemos comenzado por desarrollar un modelo sencillo en su versión no distribuida, descrito en [3] y explicado en la Sección 2.3.3 sobre los LRMM. Dicho algoritmo ha sido implementado sobre Python 2.7. Aquí, generamos los datos artificialmente con dos regresiones diferentes, teniendo así consciencia en todo momento de los parámetros 'verdaderos' de la distribución.

Lo primero que hay que tener en cuenta es que, en lugar de la suma de las logverosimilitudes, utilizamos la media de la logverosimilitud (Mean Log-Likelihood, MLL) para calcular la convergencia del algoritmo. Ésta es la suma de todas las logverosimilitudes divididas entre el número total de puntos, haciendo así que ésta sea una característica que se mantiene prácticamente invariante respecto al tamaño de nuestro conjunto de datos. Si no lo hiciéramos, la logverosimilitud sería mayor a medida que aumentamos el tamaño de los datos. Así, no habrá que modificar el umbral de convergencia en función del tamaño de nuestro conjunto de datos.

Esta MLL es un dato computacionalmente costoso de calcular. Dicho cálculo se muestra en la ecuación 39.

$$\ln p(t|\theta) = \frac{1}{N}\sum_{n=1}^{N} \ln\Big(\sum_{k=1}^{K} \pi_k \mathcal{N}(t|\omega_k^T \phi(x_n), \beta^{-1})\Big) \tag{39}$$

sin embargo, podemos darnos cuenta que ya hemos tenido que calcular previamente las verosimilitudes totales para cada término - es decir, el término que hay dentro del paréntesis del logaritmo - en el cálculo de las responsabilidades. Si nos fijamos en la Ecuación 31, vemos que el denominador es exactamente el mismo término. Así, si en el cálculo de las responsabilidades devolvemos también dichos resultados, para calcular la logverosimilitud, simplemente tendremos que promediar los resultados tras aplicar los logaritmos, consiguiendo así la mencionada MLL.

Quitando esta particularidad y la inicialización de los parámetros, de la cual se hablará más adelante en este trabajo, el resto del desarrollo modelo es prácticamente igual al mostrado en la Sección 2.3.3. Antes de nada, se selecciona un umbral de convergencia y un número máximo de iteraciones, deteniéndose el algoritmo cuando alguna las condiciones se deja de cumplir. Es decir, se detiene o bien cuando el incremento de la logverosimilitud es menor al umbral marcado y se considera que el algoritmo ha convergido, o bien cuando se alcanza el número máximo de iteraciones. Tras esto, se inicializan los parámetros de la distribución ω, π y β. Se calculan ahora las responsabilidades y se comienza el algoritmo iterativo EM, implementado con un simple *while*. En él, se calculan los parámetros, nuevamente las responsabilidades y, por último, la MLL, y se incrementa en uno el contador de

iteraciones. Tras ellos, se comprueban las condiciones del bucle, repitiéndose éste hasta que al menos una de las dos se deje de cumplir.

Los parámetros obtenidos son los que daremos como buenos para nuestro modelo. Una vez llegados a este punto, para cada dato nuevo de entrada x podremos estimar un valor de salida t. Pero, ¿cómo sabemos qué componente hemos de seleccionar para estimar el valor de salida? Aquí, a diferencia de los datos que hemos utilizado para entrenar, no tenemos información sobre el valor t, ya que es precisamente el valor que queremos estimar.

La solución aquí utilizada consiste comparar y ver cómo de parecidos son nuestros nuevos datos de entrada a los que hemos utilizado a la hora de entrenar el modelo, ordenarlos de menor a mayor por su similitud con el nuevo dato de entrada - utilizando la distancia euclídea - y obteniendo la información que necesitamos de los datos más parecidos a nuestros datos de entrada. Al tomar datos lo más similares posibles a nuestro nuevo dato de entrada, podemos asumir que su comportamiento será similar a estos. Por ello, mirando las responsabilidades de dichos datos podemos hacernos una idea de qué componente hemos de elegir a la hora de efectuar la regresión.

Una posibilidad es tomar un número determinado de datos, quedarnos para cada uno de ellos con la componente que tenga una responsabilidad mayor y elegir la componente que tenga mayoría. Sin embargo, esto podría provocar casos en los que varias componentes empatan - excepto en el caso de tomar un número impar de datos y sólo dos componentes. Un cálculo más sofisticado consiste en, con los mismos datos que hemos tomado, sumar las responsabilidades por cada componente, teniendo así un valor real por cada componente. Tras ello, dividimos los valores obtenidos por el número de datos que hemos utilizado - haciendo así que la suma de todos los valores sea 1. Por último, efectuamos una regresión para cada uno de los componentes, ponderando el valor obtenido de la regresión por el valor obtenido de las responsabilidades, y sumamos los resultados.

Así, primero estamos obteniendo un dato que puede interpretarse como una estimación de las responsabilidades del nuevo dato de entrada - que, como ya hemos visto, no se pueden calcular sin el valor de salida t -, luego llevamos a cabo todas las regresiones posibles y, para obtener el valor final, ponderamos cada valor obtenido por las estimaciones de las responsabilidades y los sumamos todos, ya que para una mayor responsabilidad, supondremos que el efecto de dicha componente sobre el dato de salida será mayor.

$$\widehat{t} = \sum_{k=1}^{K} \hat{\phi}_k(\omega_{k0} + \sum_{d=1}^{D} \omega_{kd} x_d) = \sum_{k=1}^{K} \hat{\phi}_k < x_{ext}^T, \omega_k > \tag{40}$$

donde D es el número de dimensiones de los datos de entrada, $\hat{\phi}$ es la estimación de las responsabilidades para el dato de entrada y x_{ext} es el vector extendido

de x, en el cual el primer valor es un 1, que al realizar el producto escalar irá multiplicando al valor del sesgo, la componente ω_{k0}, y se añade por este mismo motivo al inicio del vector, para poder simplificar la operación y ponerla en forma de producto escalar.

Observamos que los valores de β y de π sólo sirven para calcular las responsabilidades a la hora de seleccionar a qué componente pertenece la nueva observación y no tienen relevancia a la hora de llevar a cabo la regresión para nuevos datos. Desde un punto de vista del modelo, también nos ayudan a interpretar el mismo, viendo cómo se distribuyen los datos a través de las diferentes componentes - o lo que es lo mismo, el tamaño relativo de éstas - con la ayuda de π y viendo cómo de dispersos están los datos entre sí en el caso de β, lo que nos ayudará a contextualizar el resultado de nuestro modelo.

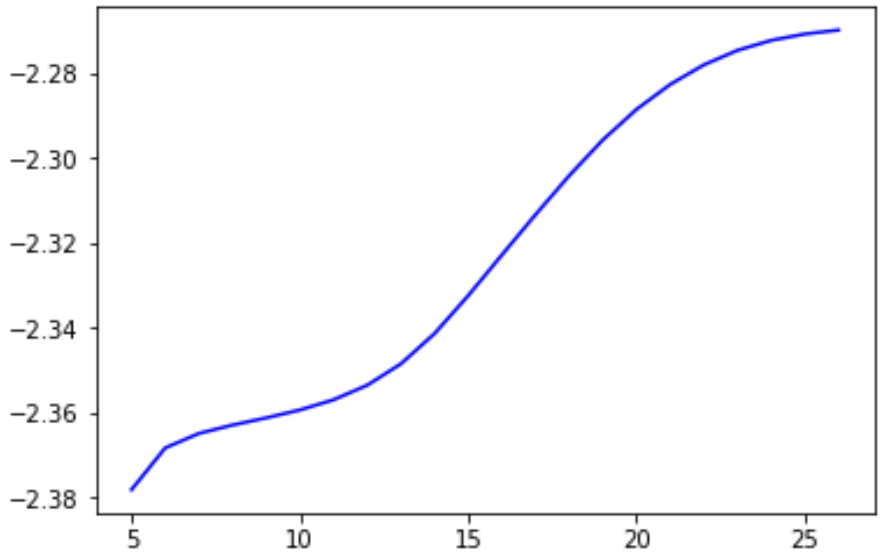
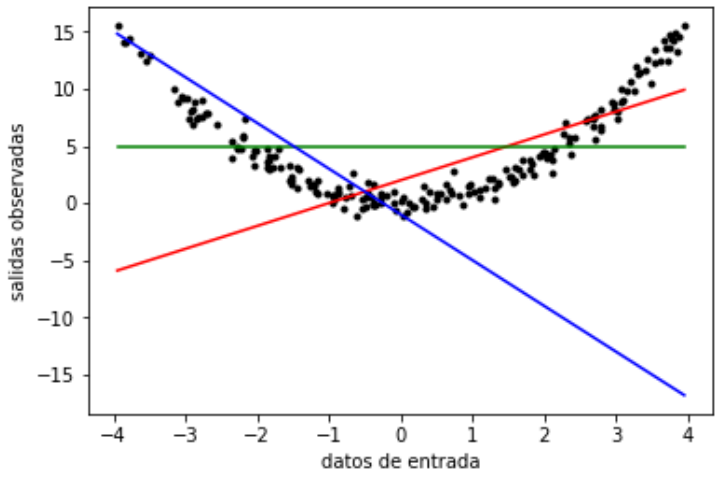


Figura 13: Ejemplo de evolución de la MLL para el ejemplo mostrado en la Figura 12.

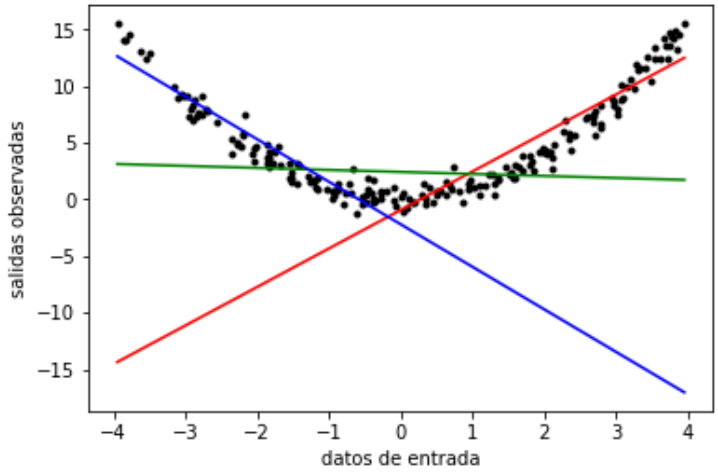
3.2. Implementación del algoritmo distribuido

Sin embargo, como ya hemos mencionado anteriormente, el algoritmo que hemos desarrollado anteriormente no se puede utilizar en distribuido. El problema reside en que, para calcular los pesos ω en el paso M de cada iteración, necesitamos efectuar operaciones con matrices, como se ve en la Ecuación 36. Estas operaciones no son implementables en distribuido, ya que para ello requerimos de todo el conjunto de datos - es decir, la matriz en su totalidad - y no podemos realizar particiones de éste, imposibilitando el trabajo en distribuido de los diferentes trabajadores.

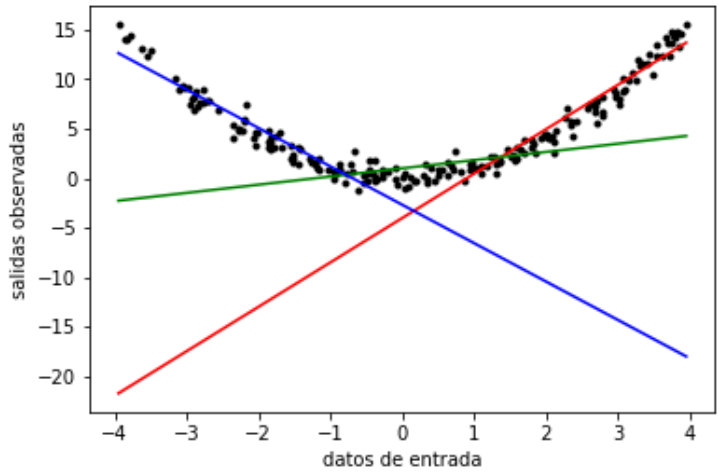
Para ello, sustituimos la solución inicial al problema de optimización de los pesos de las diferentes componentes y minimizamos en su lugar con un algoritmo de



(a) valores iniciales del modelo



(b) modelo tras 5 iteraciones del algoritmo



(c) modelo tras 20 iteraciones del algoritmo

Figura 12: Ejemplo de evolución del LRMM con tres componentes.

descenso por gradiente, explicado en la Sección 2.2. Podemos fijarnos en la Ecuación 2 para darnos cuenta de que este algoritmo si es posible de implementar en distribuido. Para calcular el gradiente necesitamos derivar la función a optimizar, que consta del termino regularizador y de un promedio de la función de costes para cada uno de los datos de nuestro conjunto. Así, cada nodo de nuestro clúster puede calcular el subgradiente la función error cuadrático - en este caso concreto, aunque se podrían utilizar otras funciones similares - para cada dato en su partición con un *map*() y sumar todos los resultados parciales obtenidos de los trabajadores con un *reduce*() y finalmente promediar por el número de puntos, con un esquema similar al mostrado en la Figura 3.

Así, desarrollando las fórmula 34, en la que tenemos el gradiente de la función de costes a utilizar en el término derecho de la ecuación, y 1, que es la función que actualiza los pesos con el método GD, obtenemos que

$$\omega_k^{n+1} = \omega_k^n - c\nabla f(\omega_k^n) = \omega_k^n - \nabla \frac{c}{N} \sum_{i=1}^{N} C(\omega_k^n; x_i, y_i) = \omega_k^n - \frac{c}{N} \sum_{i=1}^{N} \gamma_{ik}(t_i - \omega_i^{nT}\phi_i)\phi_i \tag{41}$$

Así mismo, el método SGD es igualmente posible de implementar en distribuido, ya que para cara iteración se necesita únicamente un dato para calcular el subgradiente. Desarrollando la ecuación anterior de tal forma que sólo utilizamos una observación en cada iteración, obtenemos el siguiente resultado

$$\omega_k^{n+1} = \omega_k^n - c\nabla f(\omega_k^n) = \omega_k^n - c\nabla C(\omega_k^n; x_i, y_i) = \omega_k^n - c\gamma_{ik}(t_i - \omega_i^{nT}\phi_i)\phi_i \tag{42}$$

donde en ambas ecuaciones hemos utilizado c para referirnos al SS para distinguirla de la γ, notación utilizada para las responsabilidades a lo largo de todo el trabajo.

Como ya se mencionó en la Sección 2.2, en función de nuestro conjunto de datos y los objetivos que tengamos, será mejor utilizar un algoritmo u otro. Para conjuntos muy grandes es más óptimo usar el descenso por gradiente estocástico, ya que - independientemente del tamaño del conjunto de datos - toma una única observación para cada iteración, mientras que el descenso por gradiente tradicional es más preciso y - siempre que el SS sea adecuado - siempre va a converger a una solución mejor a cada nueva iteración.

Sí utilizáramos el algoritmo SGD, tendríamos que tener en cuenta que no siempre cada paso de la iteración nos entregará una solución mejor que la anterior, como se muestra en la Figura 7, ya que no siempre el subgradiente nos llevará en la dirección o con la magnitud correctas.

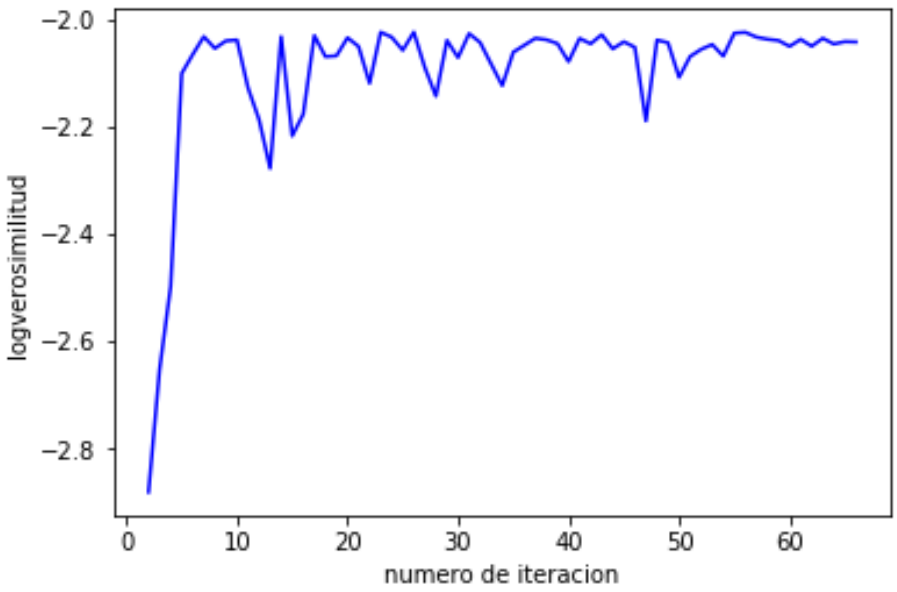


Figura 14: Evolución de la logverosimilitud utilizando descenso por gradiente estocástico.

Como ya hemos dicho, al estar trabajando en un clúster de ordenadores, será prácticamente igual de costoso calcular el subgradiente con un sólo dato - como es el caso del SGD - que con un dato por cada ordenador del mismo haciendo que el tamaño del lote del MBGD coincida con el número de ordenadores del clúster. Esto hace que nuestro subgradiente se aproxime a cada iteración a la solución con casi total probabilidad, no como en el caso del SGD. Además, el tamaño del paso no será tan crítico aquí como en el caso del SGD, y tendremos un parámetro menos del que preocuparnos, necesitando ahora únicamente que no sea ni demasiado grande ni demasiado pequeño, como en el caso del GD, pero permaneciendo constante.

3.3. Inicialización de los pesos

Como se ha mencionado anteriormente, la inicialización es crítica, sobre todo en el caso de los pesos ω de las regresiones. El hecho de tener una buena inicialización puede marcar la diferencia entre obtener una buena solución y otra no tan buena que, sin embargo, sigue siendo un mínimo local. El algoritmo no ha cambiado, simplemente lo han hecho los valores iniciales, y esto a provocado la convergencia a una mala solución.

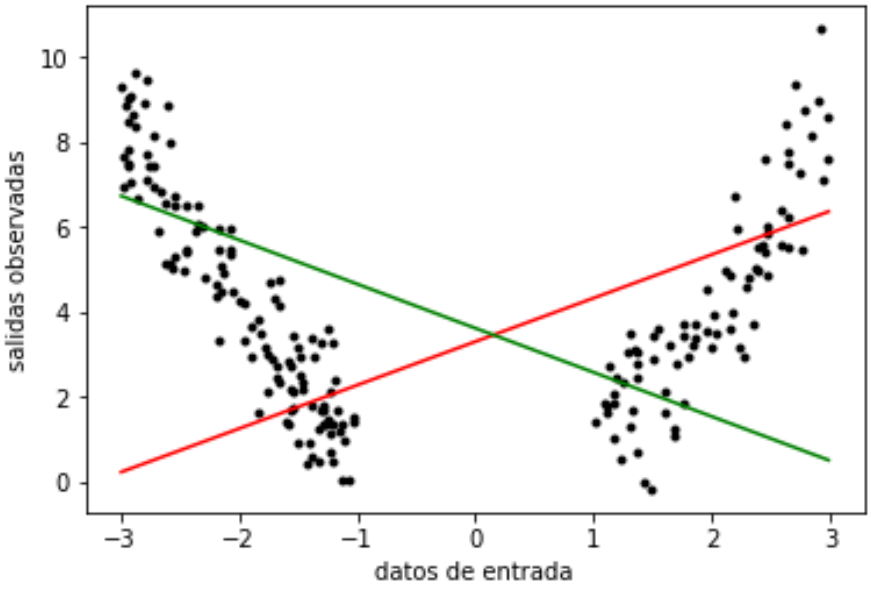


Figura 15: Ejemplo de una mala solución producto de una mala inicialización.

También el tiempo de convergencia que nos ahorra una buena inicialización puede ser crítico. Una mala inicialización puede provocar cientos de iteraciones innecesarias del algoritmo antes de converger a un mínimo, que para conjuntos de datos masivos puede llegar a suponer tiempos de convergencia críticamente superiores. Además, por mucho que la solución sea buena, hemos consumido muchos más recursos de los necesarios. Cabe destacar también que este tipo de algoritmos - tanto los EM como los GD, aunque es especialmente crítico en los primeros - suelen utilizar un parámetro que marca el número máximo de iteraciones que el algoritmo puede realizar como máximo, pudiendo así detectar una mala inicialización sin necesidad de dejar que el algoritmo converja.

Una vez hemos explicado y mostrado la importancia de la inicialización, pasamos a explicar diferentes métodos de inicialización desarrollados para este trabajo, así como sus pros y contras.

3.3.1. Inicialización mediante K-medias y regresiones lineales

Aquí entra por primera vez en juego el algoritmo de K-medias descrito en la Sección 2.4. Éste será utilizado en varios algoritmos de inicialización desarrollados a continuación. Todos ellos se basan en aprovechar los parámetros que nos devuelve el algoritmo de K-medias - que además está implementado en la librería MLlib de Spark - para estimar los diferentes parámetros y hacerlo de una manera lo más óptima y menos costosa posible.

En este caso concreto, el método de inicialización consiste en llevar a cabo un agrupamiento de los datos mediante el algoritmo de K-medias con el mismo número de componentes que deseamos tener en nuestro LRMM y, una vez definidos los

diferentes grupos, llevar a cabo una regresión lineal para cada uno de ellos, teniendo en cuenta sólo los datos de dicho grupo. Para ello, generamos un RDD con las etiquetas indicando el grupo al que pertenecen, ayudándonos luego del método $filter()$ para quedarnos con los datos pertenecientes a un grupo concreto.

Para una implementación no distribuida del algoritmo esta una magnifica opción que, a priori, no tiene muchas desventajas. Sin embargo, a la hora de buscar la versión capaz de trabajar en distribuido, surge el primer problema de esta inicialización. Mientras que en su vertiente no distribuida los pesos ω de cada regresión se calculan utilizando la matriz pseudo-inversa de Moore-Penrose, esto no es posible, como ya se ha comentado, en su versión distribuida, teniendo que utilizar alguno de los métodos de optimización basados en gradiente de los que hemos visto.

Surge aquí el siguiente problema: ¿con qué valores comenzamos para calcular el gradiente e ir actualizando progresivamente? Se podría utilizar un valor inicial fijo para todas o utilizar el valor de los centroides para alguna de las componentes y esperar a que cada regresión convergiera para cada grupo, pero esto podría provocar resultados no del todo buenos - por una mala elección del SS o por no llegar a converger de manera correcta, por ejemplo - además de consumir mucho tiempo de forma casi inútil.

El tamaño π de la componente aquí será inicializado al número de datos que pertenecen al grupo dividido entre el número total de muestras, mientras que la varianza del ruido $\frac{1}{\beta}$ se calculará como el MSE entre los valores obtenidos de la estimación y los observados para los datos de nuestro grupo, sin tener en cuenta responsabilidades.

3.3.2. Inicialización mediante un K-medias extra.

Otra forma de inicializar los datos consiste en empezar nuevamente por un algoritmo de agrupamiento de K-medias, pero en este caso con una componente más que el número de regresiones lineales que deseemos en nuestro LRMM. La idea aquí es evitarnos el tener que calcular las regresiones lineales para inicializar los pesos que, como acabamos de ver, no hay forma de hacerlo de forma rápida y precisa si trabajamos en distribuido.

Una forma de definir una recta es mediante dos puntos, y es aquí donde surge la necesidad de añadir una componente extra a la hora de buscar los grupos mediante el algoritmo de K-medias. Sí queremos N regresiones lineales, entonces tendremos $(N+1)$ grupos, y por lo tanto, también centroides. Si tomamos el primer centroide y lo unimos con el siguiente más cercano ya habremos obtenido una recta. Tras ello, unimos el segundo con el tercero, y así sucesivamente hasta tener N rectas. Así, podemos tomar de estas rectas los valores de los pesos ω, evitando tener que realizar ningún tipo de regresión.

El problema aquí es bastante obvio. Antes podíamos estimar el tamaño pi de

la componente así como la varianza el ruido $\frac{1}{\beta}$, pero ahora el método que habíamos seguido antes resulta inútil, ya que no hay grupos resultantes del K-medias asociados directamente a nuestras componentes del LRMM. Esto se puede solucionar llevando a cabo una nueva inicialización como la explicada en 3.3.2 con N componentes, pero manteniendo los valores de los pesos obtenidos en el paso anterior. Así, del primer paso obtenemos los pesos que se adaptarán a la distribución de los datos a lo largo de nuestro espacio vectorial y del segundo, la varianza del ruido y los tamaños de las componentes de forma sencilla.

3.3.3. Inicialización mediante K-medias dobles

Este método es muy similar a los dos anteriores, y vuelve a estar basado en el algoritmo de agrupamiento de K-medias. En este caso, tomamos el doble de componentes para el algoritmo K-medias que regresiones lineales queremos para nuestra mezcla. Así, los tamaños de las componentes se calcularán de forma similar al método descrito en la Sección 3.3.2, con la diferencia de que ahora habrá dos componentes del K-medias asociadas a cada componente de nuestro LRMM y habrá que tomar el tamaño de ambos grupos y dividirlo entre el total de muestras. También es análogo el cálculo del ruido, hallando el MSE para las observaciones pertenecientes a cualquiera de los dos grupos asociados a nuestra regresión.

Para ello, obviamente, hemos de calcular primero los pesos para así poder hallar los errores cuadráticos. Aquí, emparejamos los centroides por cercanía entre ellos, quedando éstos emparejados dos a dos. Tras ello, calculamos la recta para cada par de centroides que hemos obtenido del paso anterior y, de ella, obtenemos los pesos de la regresión.

3.3.4. Inicialización mediante GMM

En este apartado, nuestro método de inicialización de los parámetros está basado en otro algoritmo de agrupamiento ya explicado en la Sección 2.3.2, los GMM. El hecho de haber explorado el uso de los GMM para inicializar nuestro modelo viene motivado por el hecho de que nos aporta información extra que el K-medias no hace: por un lado, la matriz de covarianzas Σ de cada componente, y por otro, los pesos π de cada componente - con lo que dicho parámetro puede ya ser inicializado con el resultado obtenido del GMM.

Como hemos visto en otros métodos de inicialización, una de las claves es evitar tener que calcular los pesos de la regresión con algoritmos de descenso por gradiente, siendo mejor buscar 'atajos', como conseguir pares de puntos para calcular la recta y los pesos asociados a ella. En este caso, tenemos la ventaja de contar con la matriz de covarianzas. La matriz de covarianzas nos aporta información sobre en qué dirección se extienden más los datos. Así, si calculamos los autovalores y autovectores de dicha matriz, podemos obtener la dirección de mayor varianza si tomamos el autovector asociado al autovalor mayor. Así, podemos definir la recta con la media μ de la componente - parámetro que nos devuelve el GMM - y el

autovector obtenido, teniendo inicializados los pesos.

Queda por último inicializar el valor de la varianza del ruido. En métodos anteriores, requeríamos de un RDD con los datos etiquetados para realizar un cálculo explícito del MSE para cada componente. En este caso, podemos ahorrarnos dicho cálculo si volvemos a aprovechar la información que nos otorgan los autovalores de la matriz de covarianzas. Como ya hemos explicado, el autovalor más grande irá asociado al autovector que será la dirección preferente en la que se extenderán los datos, y por lo tanto esta dirección se considerará la de la propia regresión. Pero, ¿qué interpretación podemos tomar de los demás autovalores? El resto de autovalores nos da la magnitud de la varianza en otras direcciones. Por ello, en lugar de calcular explícitamente el ruido, podemos interpretar cada uno de los otros autovalores como la magnitud del ruido en cada uno de los otros ejes perpendiculares a la recta de la regresión y estimar el valor de la varianza como se muestra a continuación.

$$\beta^{-1} = \sqrt{\sum_{i=0}^{D-1} autovalor_i^2} \tag{43}$$

donde D es el número de dimensiones de nuestro problema. Ésta estimación presenta varios problemas, como el hecho de calcular el ruido con las componentes perpendiculares a la dirección de la regresión lineal y no como la diferencia entre el valor estimado $\widehat{t}$ y t. Sin embargo, el ahorro del cálculo de las etiquetas y los errores cuadráticos para cada punto y cada componente, el hecho de que el parámetro β^{-1} apenas afecta a la evolución del modelo y que es un valor que evolucionará a lo largo del LRMM hasta tomar un valor más próximo al real, hacen de esta una opción más que interesante para inicializar dicho valor.

3.4. Métodos de muestreo para la optimización del SGD y el MBGD.

A la hora de desarrollar nuestro MBGD tenemos mucha más flexibilidad que a la hora de implementar un descenso por gradiente estándar. En el algoritmo GD estándar, lo único en lo que tenemos que centrarnos es en el SS, mientras que en el caso del MBGD, el hecho de tener que muestrear nos da cierta flexibilidad para buscar cómo llevar a cabo dicha acción para obtener mejores resultados y ver las ventajas y desventajas de cada una.

Un MBGD normal puede tomar varias estrategias: muestrear con reemplazo, tomar datos hasta haberlos utilizado todos para asegurar que tomamos todas las muestras del conjunto de datos, barajar los datos antes de tomar datos sin reemplazo para evitar comportamientos cíclicos... Sin embargo, todo lo que podemos hacer aquí está relacionado con cómo tomamos las muestras.

Sin embargo, en nuestro caso tenemos una peculiaridad más a la hora optimizar la función, y esta no es otra que las responsabilidades. Un dato podría generar un subgradiente muy grande sobre determinada componente, pero este podría verse reducido hasta el punto de ser despreciable si la responsabilidad es lo suficientemente pequeña. Por ello, aquí tenemos nuevamente un abanico de posibilidades a la hora de optimizar el tiempo de convergencia del algoritmo. Algunos de los métodos que se han probado son:

- Muestrear con probabilidad de ser seleccionado proporcional a la responsabilidad de dicho punto para una componente dada.
- Marcar un umbral y tener en cuenta a la hora del muestreo sólo las observaciones cuyas responsabilidades para la componente dada sean mayores a éste.

El primero se basa en muestrear con una probabilidad proporcional a la responsabilidad de cada muestra. Para ello, tomamos la cantidad equivalente de puntos de la componente dada - del mismo modo que se hace para calcular el tamaño relativo de los grupos -, dividir la responsabilidad entre el valor obtenido y multiplicar el resultado por el número de datos que queremos tomar para cada iteración del MBGD en promedio, obteniendo así la probabilidad de cada punto de ser muestreado.

Merece la pena explicar aquí el método seguido para determinar si una muestra es tomada o no para el lote del MBGD. Primero, generamos una variable aleatoria uniforme con valores comprendidos entre 0 y 1 y tomamos una muestra. Tras ello, comprobamos si el valor obtenido se encuentra por debajo del valor obtenido anteriormente. Así, el área comprendida entre 0 y la probabilidad del dato de ser muestreado es exactamente dicha probabilidad, como se muestra en la figura 16

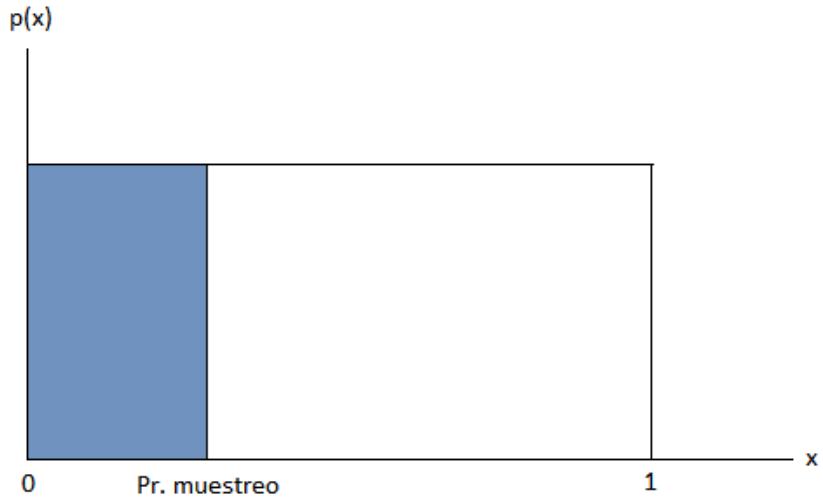


Figura 16: Probabilidad de muestreo de un dato. El área coloreada coincide con la probabilidad calculada anteriormente.

El problema aquí reside en que el total de muestras tomadas en cada iteración variará, siendo a veces inferior y a veces superior al valor deseado, pero a cambio estamos tomando muestras estadísticamente más significativas. Aquí, el tamaño del lote elegido será el valor esperado del número de muestras en cada iteración.

En el segundo caso, simplemente despreciamos aquellos datos cuya responsabilidad para la componente dada esté por debajo de un umbral determinado, eliminando la posibilidad de muestrear datos que apenas influirían en el subgradiente de haber sido muestreadas, ahorrando así cálculos innecesarios y que ralentizan el proceso iterativo del algoritmo.

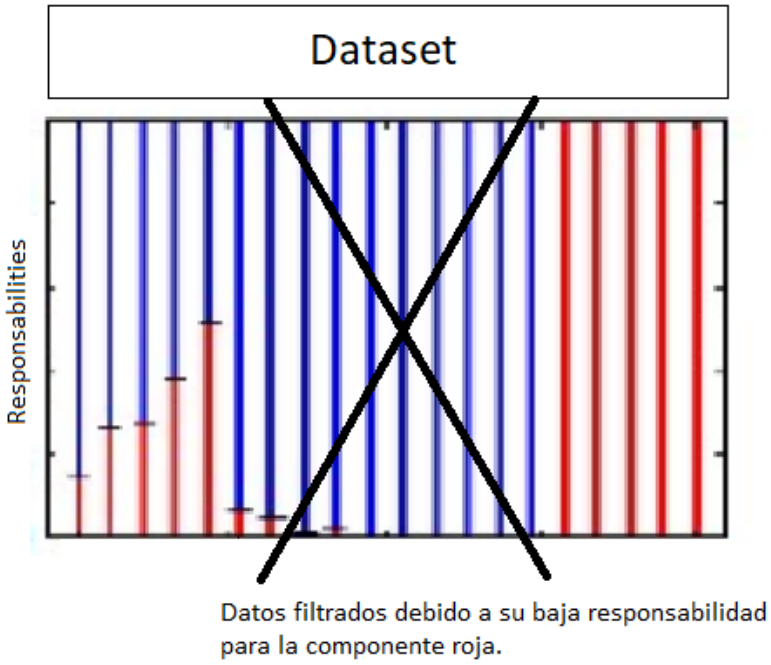


Figura 17: Filtrado de los datos con una responsabilidad muy baja para evitar su muestreo en el MBGD.

La implementación de este método de muestreo es muy sencilla. Para una componente dada, se filtra el RDD para quedarnos sólo con aquellas responsabilidades por encima del umbral y después se procede a muestrear. Sin embargo, el segundo caso presenta más problemas.

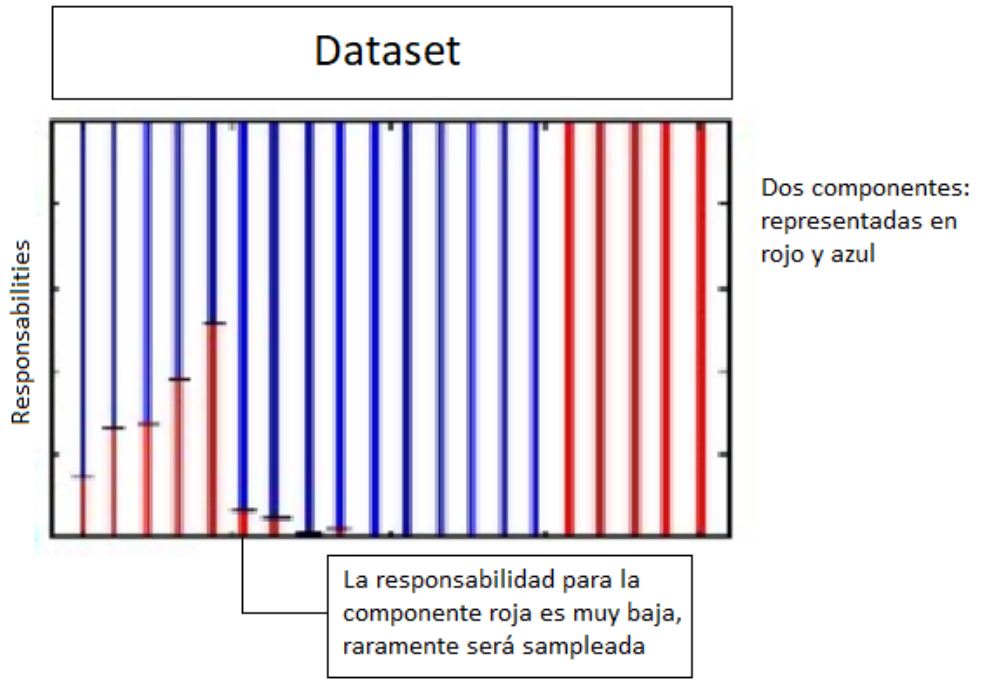
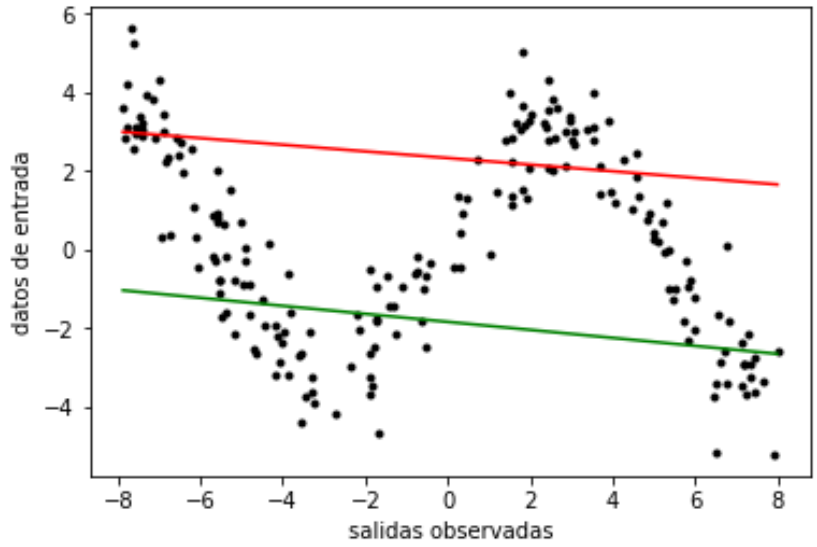


Figura 18: Responsabilidades de las muestras de un LRMM con dos componentes. Haremos un muestreo con probabilidad proporcional a la responsabilidad.

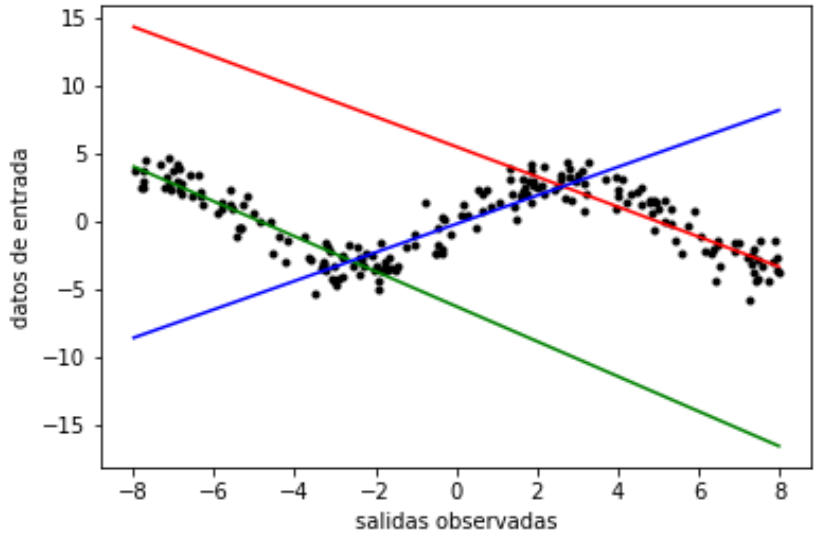
3.5. Selección del número de componentes

Otro tema importante es una buena selección del número de componentes. Está claro que la elección de un número de componentes óptimo dependerá de nuestro problema. Al utilizar regresiones lineales simples, requeriremos de un número mínimo de componentes para poder adaptar nuestro modelo al problema con unas mínimas garantías, como se muestra en la Figura 19, en la que vemos que requerimos de tres componentes para obtener una solución aceptable a nuestro problema.

Sin embargo, la elección de un número mayor al que se puede considerar óptimo no tiene, en general, mayor repercusión en el modelo, como se muestra en la Figura 20. Tendremos más componentes y cada una de estas se adaptará a una porción más pequeña de los datos, pero lo seguirá haciendo de forma correcta. Aunque es cierto que no siempre el número de componentes, si este es mayor que el necesario, se va a seguir adaptando bien al problema, el mayor problema aquí reside en el coste que supone aumentar el número de componentes, ya que la carga se verá aumentada en proporción al número de éstas. Además, elegir un número de componentes mayor al necesario afectará especialmente al resultado en casos en las que la distribución del problema tenga discontinuidades.



(a) Tenemos menos componentes de las necesarias y el LRMM converge a una mala solución.



(b) Tenemos un número correcto de componentes, convergiendo el LRMM a una buena solución.

Figura 19: Ejemplo de una buena y mala elección del número de componentes del LRMM.

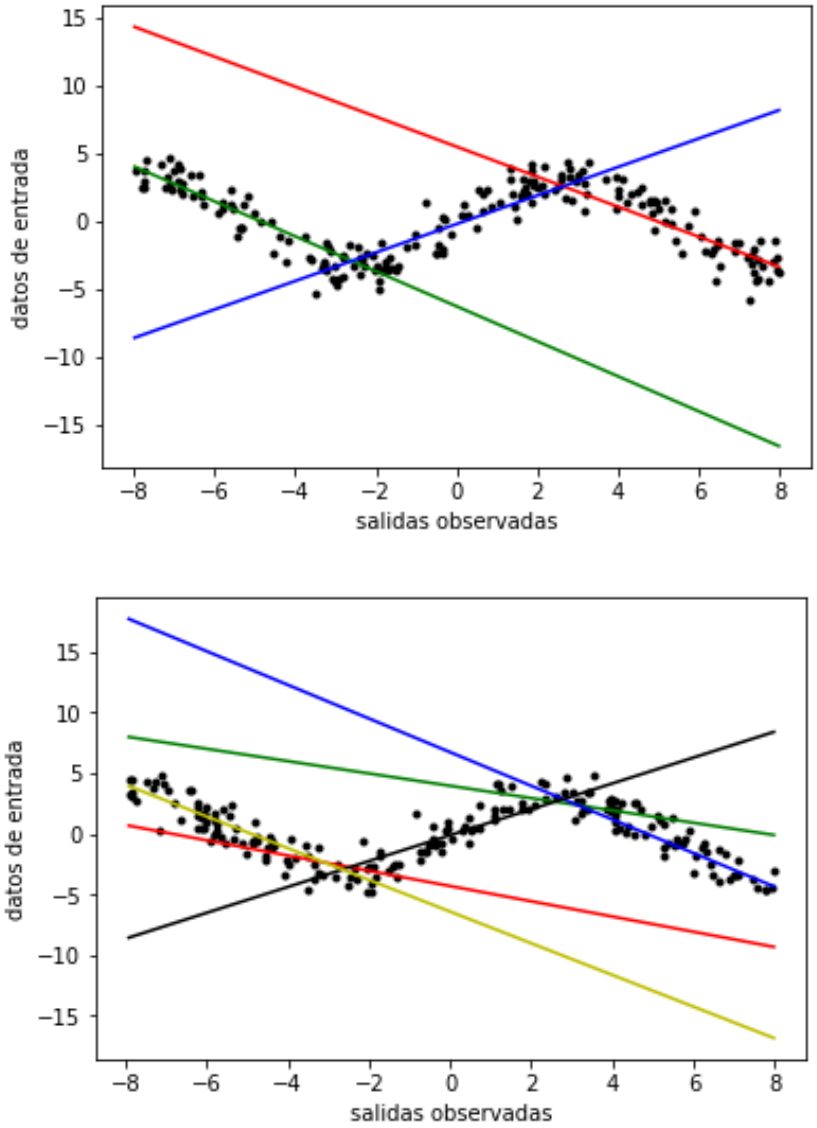


Figura 20: Ejemplo de como el modelo no empeora por elegir un número de componentes mayor que el necesario.

4. Experimentos

4.1. Conjuntos de datos y algoritmos a utilizar.

En esta sección se presentarán y enumerarán los diferentes conjuntos de datos, así como los algoritmos de optimización desarrollados y los métodos de inicialización del modelo. Además, asignando números a cada algoritmo o conjunto de datos, podemos presentar los resultados de forma más ordenada y menos confusa.

En primer lugar, presentaremos los conjuntos de datos sobre los que se llevará a cabo el experimento. Todos ellos son conjuntos de datos cuyas salidas guardan relaciones no lineales con las entradas. Se han utilizado conjuntos de 10.000 muestras cada uno para la realización de los experimentos. A continuación, se mostrará una submuestra de 100 datos de cada conjunto y se explicará el procedimiento seguido para su obtención.

- Conjunto de datos #1: consistente en dos rectas paralelas sin pendiente y diferente sesgo.
- Conjunto de datos #2: consistente en una función sigmoide.
- Conjunto de datos #3: consistente una función sinusoidal. Se extiende tres cuartos de su periodo.
- Conjunto de datos #4: consistente en el valor absoluto del valor de entrada.

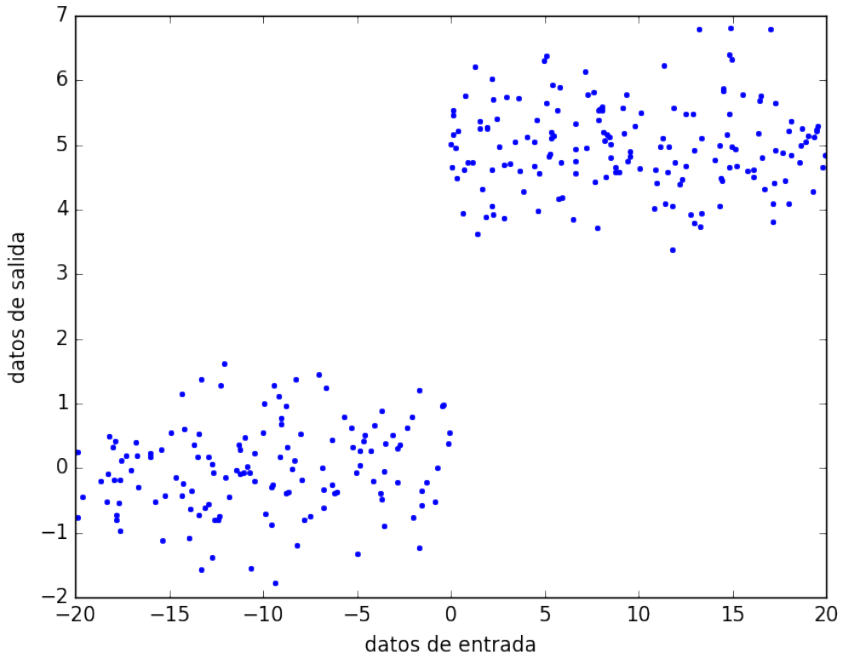


Figura 21: Conjunto de datos #1. Dos rectas paralelas en diferentes tramos.

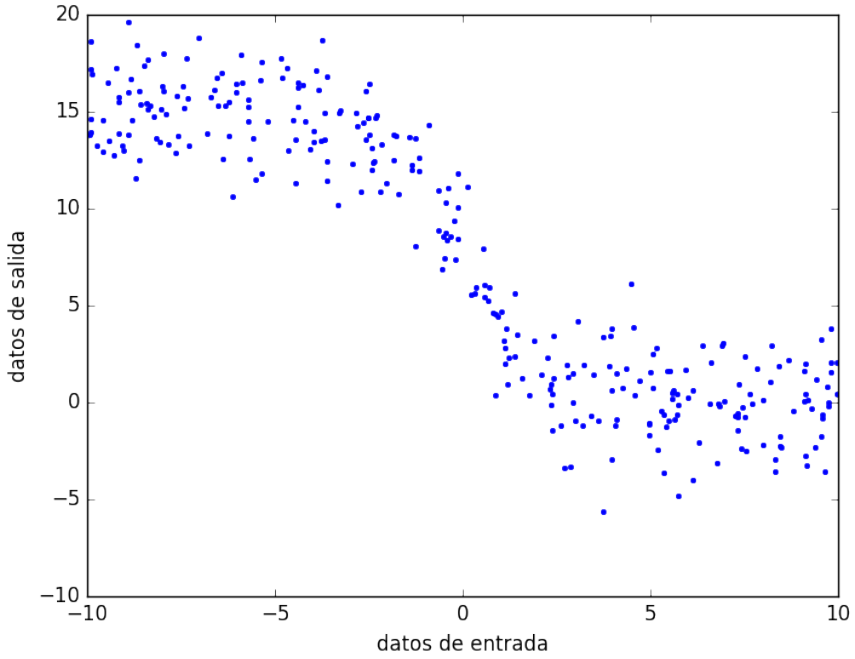


Figura 22: Conjunto de datos #2. Función sigmoide.

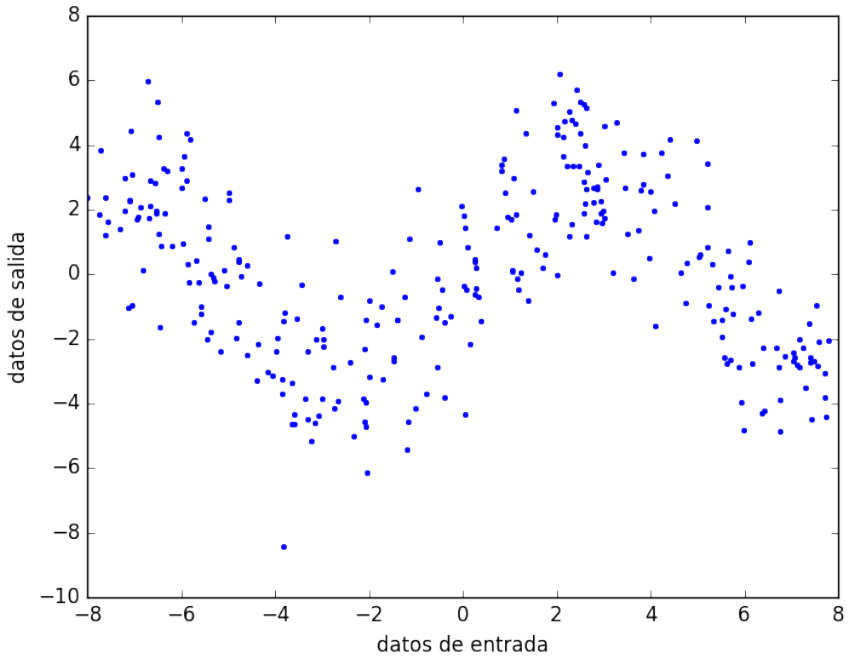


Figura 23: Conjunto de datos #3. Tres cuartos del periodo de una función sinusoidal.

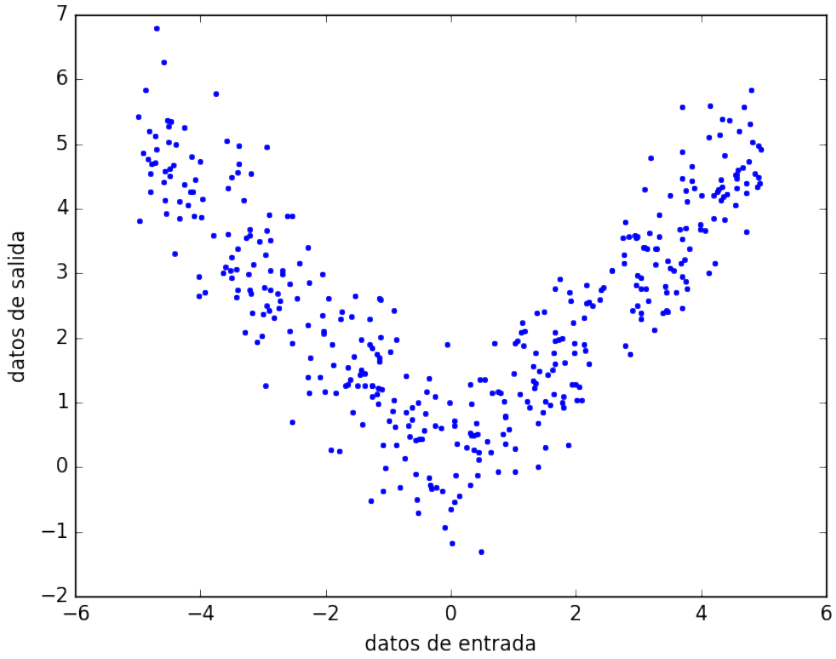


Figura 24: Conjunto de datos #4. Valor absoluto.

Además, todos los conjuntos de datos tienen ruido aditivo Gaussiano de media nula. Se ha buscado utilizar conjuntos de datos que pongan a prueba diferentes partes de este trabajo, como la inicialización de los parámetros o las convergencias a buenas soluciones. En estos experimentos, se ha utilizado un número de componentes suficiente como para adaptarse bien a cada uno de los problemas, siendo 2 componentes para los conjuntos de datos #1 y #4 y 3 componentes para #2 y #3.

Pasamos ahora a describir los diferentes algoritmos de inicialización con su respectiva enumeración.

- Inicialización #1: inicialización mediante K-medias y regresiones lineales.
- Inicialización #2: inicialización mediante un K-medias extra.
- Inicialización #3: inicialización mediante K-medias dobles.
- Inicialización #4: inicialización mediante GMM.

Y por último, los diferentes algoritmos de optimización utilizados.

- Optimización #1: optimización mediante MBGD estándar.
- Optimización #2: optimización mediante filtrado de las responsabilidades.
- Optimización #3: optimización mediante muestreo con probabilidad proporcional a las responsabilidades.

Además, para una mayor claridad, utilizaremos las abreviaturas 'Init' para inicialización, 'Opt' para optimización y 'DS' para los conjuntos de datos en las tablas de resultados.

4.2. Tiempos de convergencia.

En esta sección, se medirán los tiempos de convergencia de los diferentes algoritmos, tanto de optimización como de inicialización de los parámetros desarrollados a lo largo de este TFG. En el caso de los algoritmos de optimización, se medirán todos con la misma inicialización y semilla, para estar en igualdad de condiciones mientras que para las inicializaciones se tomarán dos medidas: el tiempo que tarda la propia inicialización en llevarse a cabo y el tiempo que tarda en converger el modelo con cada una de ellas una vez inicializado. Para este caso, se utilizará el algoritmo MBGD tradicional como algoritmo de optimización de los pesos.

En la Tabla 1, se encuentran los tiempos relativos a las inicializaciones. En las columnas se encontrarán las medidas para los diferentes conjuntos de datos. En las filas, las inicializaciones, habiendo dos filas por cada algoritmo de inicialización: en una de ellas irá solamente el tiempo de inicialización (indicado como 'Init') y en la otra el del convergencia del algoritmo tras ser inicializado (indicado como 'LRMM').

Hay que tener en cuenta también que los tiempos de ejecución son muy variables. Por ello, se han realizado 5 mediciones de cada tiempo y tomado el valor medio, excepto en los caso en los que el tiempo era lo suficientemente grande como para poder sacar conclusiones sin necesidad de repetir el experimento más veces.

Tabla 1: Tiempos de convergencia en segundos de los métodos de inicialización.

		DS #1	DS #2	DS #3	DS #4
Init #1	Init	58,233	92,620	89,637	70,008
Init #1	LRMM	74,284	95,095	125,530	16,397
Init #2	Init	5,612	17,326	12,335	8,231
Init #2	LRMM	5,514	25,912	12,815	18,634
Init #3	Init	4,915	15,347	7,282	6,979
Init #3	LRMM	5,223	17,373	13,556	11,434
Init #4	Init	4,028	6,947	5,818	4,041
Init #4	LRMM	3,077	42,139	15,957	10,148

En la Tabla 2, se encuentran los tiempos de convergencia para los diferentes algoritmos de optimización. Para ello, se han tomado inicializaciones subóptimas aposta - la misma para todos los algoritmos -, para que el algoritmo requiriera de un mayor número de iteraciones, pudiendo así medir mejor las diferencias en los tiempos.

Tabla 2: Tiempos de convergencia en segundos de los algoritmos de optimización.

	DS #1	DS #2	DS #3	DS #4
Opt #1	5,132	14,875	14,320	10,176
Opt #2	6,566	22,842	23,552	16,037
Opt #3	12,418	56,413	46,939	30,920

4.3. Prestaciones del modelo.

Para medir la efectividad del modelo para sus diferentes inicializaciones y algoritmos de optimización utilizaremos el MSE. Además, en última instancia se comparará con una SVM con Kernel Gaussiano, para poder comparar los resultados con un algoritmo puntero en la resolución de problemas de regresiones no lineales.

En la Tabla 3, cada columna representa un conjunto de datos diferente, mientras que las filas muestran diferentes tipos de inicialización. En este caso, se utiliza el algoritmo de MBGD para optimizar los pesos en todos los casos. En la Tabla 4, las columnas siguen siendo los diferentes conjuntos de datos, mientras que las filas representan los diferentes algoritmos de optimización. Se utilizarán aquí inicializaciones que se adapten mejor a cada conjunto de datos. Se añadirá aquí una fila con los datos obtenidos por la SVM.

Tabla 3: Prestaciones de las diferentes inicializaciones.

	DS #1	DS #2	DS #3	DS #4
Init #1	2,7384	12,5530	5,6293	4,3242
Init #2	2,2689	5,6626	7,4320	0,5698
Init #3	0,8893	5,3972	3,5097	0,5180
Init #4	0,8905	5,5745	3,9671	0,5252

Tabla 4: Prestaciones de los algoritmos de optimización.

	DS #1	DS #2	DS #3	DS #4
Opt #1	0,8890	5,3972	3,9711	0,5252
Opt #2	0,8903	5,5950	3,9685	0,5253
Opt #3	0,8893	5,5737	3,9679	0,5252

Tabla 5: Prestaciones de un SVM con Kernel Gaussiano.

	DS #1	DS #2	DS #3	DS #4
SVM	0,8894	5,1853	3,5102	5,1856

5. Conclusiones y futuras líneas de investigación

5.1. Conclusiones.

En primer lugar, hablaremos de los algoritmos de inicialización. Tomando como base el algoritmo de inicialización #1, basado en agrupamiento mediante K-medias y realizar una regresión lineal, vemos que el resto de enfoques lo superan tanto en prestaciones como, sobre todo, en tiempo de convergencia. Aquí, la clave reside en la lentitud y poca precisión de las regresiones lineales implementadas en Apache Spark - aunque también se ha probado con implementaciones propias, con resultados aún más lentos - sin tener información previa sobre la distribución de los datos. El resto de algoritmos de inicialización obtienen mejores resultados tanto en prestaciones como en tiempos de convergencia, debido a que estimamos la dirección de los datos - que usaremos para estimar los pesos de las regresiones - sin utilizar métodos de descenso por gradiente.

En cuanto a prestaciones, vemos que los mejores algoritmos son los números #3 y #4 , siendo este último ligeramente superior, en prestaciones, pero siendo el primero más rápido en general. Además, se ve en los tiempo de convergencia del LRMM que son los valores que mejor inicializan el problema, ya que también estos tiempos son los más bajos. Respecto a los otros métodos de inicialización, vemos que el #1 es una mala inicialización para los DS #1, #2 y #3, mientras que el método #2 lo es para los DS #1 y #3, siendo esto provocado en el primer caso por las discontinuidades, a las que el algoritmo no se adapta bien, y en el segundo debido a la complejidad de los datos.

Respecto a los algoritmos de muestreo, vemos que todos tienen unas prestaciones prácticamente iguales, con lo que hemos conseguido no empeorar las prestaciones del algoritmo de MBGD estándar. Sin embargo, no se ha conseguido que los algoritmos converjan en un tiempo menor. Esto seguramente se deba a los pasos intermedios que hay que llevar a cabo para muestrear datos más significativos o eliminar los insignificantes antes de realizar el MBGD.

Además, se han comparado las prestaciones con un SVM con Kernel Gaussiano que, como ya hemos mencionado, no está todavía integrado con Spark. Vemos que los resultados del SVM son prácticamente iguales a los nuestros, siendo ligeramente superior para los DS #2 y #3 e inferior para el #4. Así, vemos que nuestro modelo obtiene resultados prometedores, comparables a algoritmos del estado del arte actual para problemas de regresión no lineales, estando además completamente integrado con Spark, habiendo pocos algoritmos de este tipo en la librería MLlib.

5.2. Futuras líneas de trabajo en investigación

En primer lugar, se debería evaluar las prestaciones del modelo con alguna base de datos real y típica de los problemas de Big Data para poder probar el modelo en un entorno más real y comprobar si sigue siendo válido en un entorno en el que los datos no sean tan artificiales. Aquí, podríamos volver a enfrentar nuestro LRMM frente a otros algoritmos del estado del arte, como las NN, y poder así comparar su efectividad frente a los algoritmos más punteros y que mejor funcionan para este tipo de problemas. También se podría indagar más en cómo afectan los diferentes parámetros de los algoritmos de nuestro modelo a los resultados del mismo, como el número de datos que tomamos cuando nos llega un nuevo dato para estimar las responsabilidades de éste, o investigando más sobre métodos de muestreo para tomar muestras significativas a la hora de efectuar nuestro MBGD para la optimización de los pesos.

En segundo lugar, el modelo se podría mejorar trabajando en conjunto con Óscar García Hinde, personal docente e investigador del Departamento de Teoría de la Señal y Comunicaciones de la Universidad Carlos III de Madríd, quien está trabajando en un desarrollo de LRMM en su vertiente no distribuida y cuyo modelo mejora en prestaciones al mostrado en [3], sobre el que se ha desarrollado en este trabajo. Así, se podrían integrar las mejoras introducidas por su modelo, adaptándolas a las necesidades del la computación distribuida, y consiguiendo unas prestaciones mejores a las del modelo aquí desarrollado.

Por último, se podría extender la formulación a problemas de clasificación para conseguir obtener modelos de mezclas de regresiones logísticas (Logistic Regression Mixture Models), para poder así probar la efectividad de este tipo de modelos en problemas de clasificación. Además, con todo el material recogido, se podría desarrollar un ToolboX - un entorno de desarrollo pensado para introducir a la programación de ordenadores sin necesidad de tener competencias informáticas [13] - para facilitar el uso del modelo.

6. Planificación y presupuesto del proyecto

En la primera parte de esta sección se justificará la planificación del proyecto así como el desarrollo de cada uno de sus subapartados. En la segunda parte, se hablará de los costes del proyecto y su presupuesto.

6.1. Planificación del proyecto

El proyecto se ha desarrollado a largo un año en el que se han llevado a cabo las diferentes etapas del proyecto, mostradas en la Figura 25. En una primera etapa se fijó el tema a desarrollar en este trabajo, así como a marcar unos objetivos iniciales y fijar unas guías para el mismo. Ésto se llevó a cabo en una reunión en persona entre el alumno y su tutora. A esta le siguió una etapa de transmisión de conocimiento por parte del tutor hacia el alumno, en la que este último había de aprender ciertos conceptos de vital importancia para el desarrollo de este trabajo. Cabe destacar la extensión e importancia de esta etapa, puesto que la mayoría de los conceptos iban más allá a los conocimientos impartidos durante la carrera.

	Tarea	Inicio	Fin
1	Planificación del proyecto	sept. 21	sept. 24
2	Transferencia y obtención de conocimientos	sept. 25	nov. 11
3	Familiarización con Apache Spark	nov. 12	ene. 15
4	Implementación GMM local	ene. 26	feb. 07
5	Implementación GMM distribuido	feb. 08	mar. 01
6	Implementación LRMM local	mar. 02	abr. 15
7	Implementación LRMM distribuido	abr. 16	may. 20
8	Desarrollo métodos de inicialización	may. 21	jun. 03
9	Desarrollo métodos de muestreo para MBGD	jun. 04	ago. 27
10	Realización de experimentos	ago. 28	sept. 8
11	Desarrollo de la memoria	sept. 09	sept. 23

Figura 25: Tareas del proyecto junto con su duración.

Así, durante casi dos meses, el principal objetivo fue el de conseguir el conocimiento necesario en el campo del ML, especialmente en GMM, la optimización mediante diferentes algoritmos basados en descenso por gradiente. Tras ello, se pasó a una etapa en la que el alumno debía de familiarizarse con Apache Spark como herramienta para el desarrollo del TFG, ya que éste no tenía conocimientos previos de dicho framework. Aquí entran tareas como aprender los fundamentos de Apache Spark para trabajar de forma distribuida en clústeres de computadores y la sintaxis específica que necesitabamos para trabajar con PySpark - la implementación de Spark sobre Python.

Para obtener el conocimiento relativo tanto al desarrollo de los GMM como a una primera aproximación de los LRMMs hemos optado por [3]. Para la parte de optimización se ha optado por [9] para la parte GD y de [10] para el muestreo

en el SGD y MBGD. También destacar la publicación [12], que van en la línea de lo explicado en la Sección 3.4 referente a la importancia del muestreo. La información relativa a Apache Spark se obtuvo con los cursos de EdX de [6] para los conceptos básicos de Apache Spark y [8] para la parte de ML, así como del libro [5].

Tras ello, se comenzó con implementaciones de GMM tanto en local como en distribuido para tener una base sobre la que empezar la siguiente etapa, desarrollar nuestro algoritmo de LRMM primero en local y luego en distribuido. A pesar de que el objetivo del trabajo fuera la implementación de un algoritmo distribuido que pudiera trabajar con datos masivos de forma distribuida en un clúster de ordenadores, se optó por comenzar por una implementación no distribuida - ya que el modelo presentado en el [3] tenía ciertas carencias. Para ello, se trabajó con Python 2.7. Tras la implementación del LRMM en local se migró el código para crear un algoritmo completamente distribuido sobre Spark.

Una vez desarrollado el modelo, se pasó al estudio de la inicialización de los parámetros del modelo, así como en realizar ciertas modificaciones en el muestreo a nuestros algoritmos de optimización de MBGD para mejorar las prestaciones del modelo.

La última etapa estuvo formada por los experimentos realizados con nuestro modelo. En ellos, se miden diferentes variables que nos darán una idea de la efectividad de nuestro modelo, tales como los tiempos de convergencia y las prestaciones de los diferentes algoritmos de inicialización y muestreo del MBGD implementados así como las prestaciones y tiempos de convergencia de nuestro modelo comparadas con una SVM con Kernel Gaussiano. Tras ello, se procedió a hacer un sumario de todo el material y la información recopilada a lo largo del proceso y, con ello, redactar la memoria del trabajo. En la Figura 26 se muestra un diagrama de Gannt con la distribución de las tareas a lo largo del tiempo.

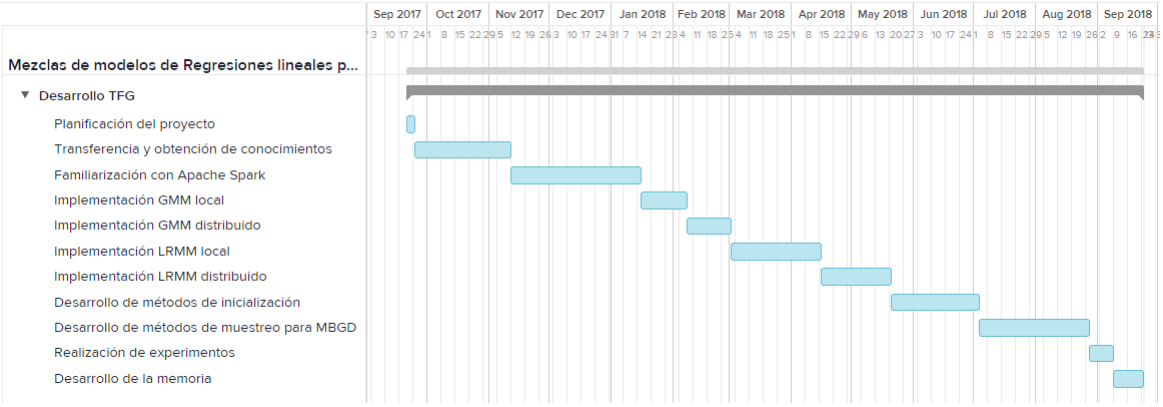


Figura 26: Diagrama de Gannt del proyecto.

6.2. Presupuesto del proyecto

En este apartado se explicarán los costes del proyecto, desarrollando el presupuesto necesario para la finalización del mismo. En un primer lugar, tendremos en cuenta los gastos personales. En este proyecto, han participado únicamente dos personas:

- El alumno, considerado aquí como un ingeniero Junior.
- La tutora, considerada como investigadora senior.

El número de horas de cada uno de los miembros implicados en el desarrollo son valores estimados y no exactos. Conociendo el coste por en función del puesto, se calculan los costes personales asociados al proyecto, que se mostrarán en la Tabla 6.

Tabla 6: Costes personales

Personal	**Horas de trabajo**	**Coste por hora (€/h)**	**Total (€)**
Ingeniero Junior	500	12	6.000
Investigador Senior	50	23	1.150
		TOTAL	**7.150**

Por otro lado, hay que tener en cuenta los gastos relativos al material, hardware y software necesarios. Aquí, al estar desarrollado sobre Software de código abierto - Python y Apache Spark -, los gastos por software son nulos, siendo el coste únicamente el asociado al ordenador de trabajo del alumno así como el clúster de ordenadores del departamento de Teoría de la Señal y Comunicaiones de la Universidad Carlos III de Madrid. Dichos gastos quedan ilustrados en la Tabla 7.

Tabla 7: Costes materiales

Concepto	**Cantidad**	**Precio (€/u)**	**Amort.**	**Total (€)**
Toshiba Satellite P850-31M	1	1.050	10 %	105
Nodos del clúster	1	18.000	1 %	180
			TOTAL	**285**

Así, sumando tanto los gastos personales como de material y añadiendo los impuestos pertinentes - el Impuesto sobre el Valor Añadido o IVA en este caso -, tenemos la Tabla 8 con el presupuesto total del proyecto.

Tabla 8: Presupuesto total

Concepto	**Total (€)**
Costes personales	7.150
Costes de material	285
Costes totales	**7.435**
IVA (21 %)	1.561,35
Presupuesto total	**8.996,35**

Referencias

[1] V. Mayer-Schönberger y K. Cukier. *Big Data: a revolution that will transform how we live, work and think*. London, John Murray, 2013.

[2] MLlib: librería de Spark para algoritmos de aprendizaje automático. `https://spark.apache.org/docs/latest/api/python/pyspark.ml.html`

[3] C. Bishop. *Pattern Recognition and Machine Learning*. Berlin, Springer Verlag, 2006.

[4] R. Duda y P. Hart. *Pattern Classification and Scene Analysis*. New York, Wiley, 1973.

[5] A. Alexander. *Scala Cookbook*. Sebastopol, CA: OReilly & Associates, 2013.

[6] EdX online courses. *Introduction to Apache Spark.*. Código: BerkeleyX - CS120x

[7] S. Suryansh. *Gradient Descent: All You Need to Know*. `https://hackernoon.com/gradient-descent-aynk-7cbe95a778da`

[8] EdX online courses. *Distributed Machine Learning with Apache Spark.*. Código: BerkeleyX - CS105x

[9] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge: Cambridge Univerisity Press, 2004.

[10] L. Bottou. *Stochastic Gradient Descent Tricks*. Lecture Notes in Computer Science Neural Networks: Tricks of the Trade, pp. 421–436, 2012.

[11] A. P. Dempster, N. M. Laird, D. B. Rubin. *Maximum Likelihood from Incomplete Data via the EM Algorithm*. Journal of the Royal Statistical Society. Series B (Methodological), Vol. 39, No. 1, pp. 1-38, 1977.

[12] D. Csiba and P. Richtárik. *Importance Sampling for Minibatches*. Journal of Machine Learning Research, vol. 19, pp. 1–21, 2018.

[13] F. Vico. *ToolboX: Una estrategia transversal para la enseñanza de la programación en entornos educativos*. ReVisión, vol. 10, no. 2, pp. 53-68, 2017.